1. INTRODUCTION

Industrial automation aims at taking over repetitive tasks. Thereby allowing workers to do high skill jobs as well as increases productivity and quality of products. This study aims to build a software library under the name fx_library for Fluxana GmbH & Co. KG with the use of Universal Robots UR3e to automate the process of loading and unloading crucibles and glass beads into the Vitriox 4+ machine. This chapter serves as an introduction to the company for which the Fluxana Library, the robot brand and the functions that will be included in the library.

1.1 FLUXANA INTRODUCTION

Fluxana was founded in 2002 by Dr. Rainer Schramm and is a registered trademark for X-ray Fluorescence (XRF) solutions. Fluxana is an accredited manufacturer of reference materials. There are several more services that Fluxana offers which include using the Vitriox 4+ machine in Figure 1.1.1 for material analysis for up to four samples at a time and up to sixteen samples in total.

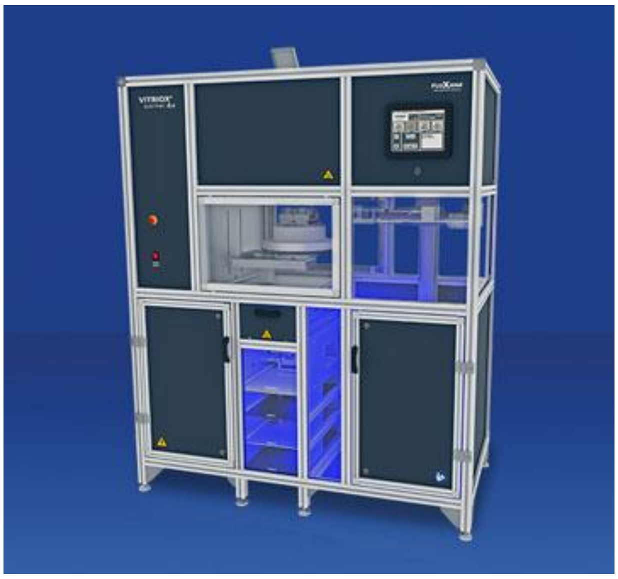


Figure 1.1.1: Vitriox Electric 4+ [92].

Currently, the samples are loaded and unloaded manually by a user, however, it is aimed to automate the process to invest human workforce into tasks that require skill rather than repetitiveness.

Starting from February 2021, Fluxana has been working with Universal Robot's UR3e to automate the loading and unloading process. Programming of the cobot was initially done through the Universal Robot API which is available on the polyscope that comes with the robot. Afterwards, Fluxana's own API was created using Visual Basic programming language and connecting to the robot with an Ethernet cable for communication over socket connections. Socket connections are links that allow communication on a network between two ends. This approach, however, was limited since all functions had to be coded from scratch. Therefore, the next step was to create the API using Robot Operating System (ROS), where already existing open-source code can be used to code improved versions of the Fluxana Library functions required for the pick and place application for the Vitrox 4+.

On the long term Fluxana plans to use artificial intelligence and computer vision to enhance the performance of the library by checking if the crucibles have been loaded correctly into the machine before turning on the furnace and starting the fusion operation. This is to avoid accidents that may occur at the high centrifugal forces inside the machine. The state of the glass beads will also be checked using machine learning models to make sure that the glass is not broken before unloading it.

1.3. MOTIVATION

The purpose of this study is to build the Fluxana Library which contains functions aimed to automate the pick and place application for the Vitriox 4+. This library will be built using Robot Operating System (ROS) to benefit from its already existing libraries and functionalities. The functions included range from ones that are used for starting up the robot, and activating external control, move the robot given different types of arguments to describe the end target and velocity and acceleration used to reach it, as well as functions to stop and shut down the robot. The scope of this study is limited to setting the speeds and accelerations for the joints and not setting them for the end effector as well as does not include functions for controlling the end effector, these functions for gripping are, however, planned to be implemented and added to the library in the future as they are essential for the pick and place application. For better understanding of some of the commands, the reader needs to have basic knowledge of Linux terminal commands.

1.4. REQUIREMENT SPECIFICATIONS

A software library is to be developed to get easy access of Cobot commands from an API software for Fluxana. The goal is to use the same concepts to work with UR3e and NIRYO Ned2 cobots. This thesis will only discuss the Fluxana Library for the UR3e. The list of functions required for the library are as follows:

- Going to home position
- Moving along an axis
- Moving to joint values
- Moving to a pose
- Moving from start to end pose
- Stopping movement
- Releasing brakes
- Powering off
- Powering on
- Getting robot mode
- Initializing the robot
- Getting remote control status
- Shutting down
- Starting external control
- Stopping external control
- Unlocking protective stop

Each of these functions will be discussed in more detail in the Fluxana Library in chapter 4.

1.5. CHAPTER DESCRIPTION

Chapter 2 will lay down the theoretical background necessary in programming the Fluxana Library and will be divided into a section about robot kinematics and one about Robot operating System (ROS). Chapter 3 will continue to describe the system architecture used in the study as well as the file hierarchy used. Chapter 4 contains a breakdown and description of the Fluxana Library. Chapter 5 contains tests done to check the performance of the Fluxana Library functions. Chapter 6 includes the conclusion of the study along with a brief description of the future works and projects Fluxana plans to do.

2. THEORETICAL BACKGROUND

It is important to understand some basic concepts regarding kinematics of robots and Robot Operating System (ROS) before going into the implementation of the Fluxana Library. Basic robot kinematics knowledge is needed in this thesis as there are move commands in the library which requires the understanding of how to give the robot commands for moving to a specified target and to be able to understand the difference between the different representations of that target. It is also important to understand what a kinematic chain is and how Denavit-Hartenberg parameters are used in describing it.

As for ROS, it is a widely used framework that allows the creation of scalable robotic applications. Knowledge about this framework is necessary as it will be used in creating the Fluxana Library. The ROS concepts discussed will range from:

- Explaining what a node is and how to launch it
- Different types of communications used by ROS
- MoveIt! Framework for motion planning and collision checking
- Rviz for visualization in robotic applications
- URDF and Xacro for robot description

Each of the mentioned concepts are explained in more detail throughout this chapter.

2.1. KINEMATICS OF ROBOTS

Robot kinematics studies the movement of the robot manipulator with regards to a reference frame without considering the forces acting on it [17]. Moving the robot to a target pose can be done linear either in joint or tool space, where:

- Joint space: describes the robot angular position of the manipulator using the six joint values of the UR3e.
- Tool space: used for moving the robot tool tip in a straight line and is defined in terms of position and orientation of the tool tip.

Figure 2.1.1 shows the coordinate system, also referred to as the base frame, used by the robot on the left and the tool center point (TCP) used for planning in tool space on the right:

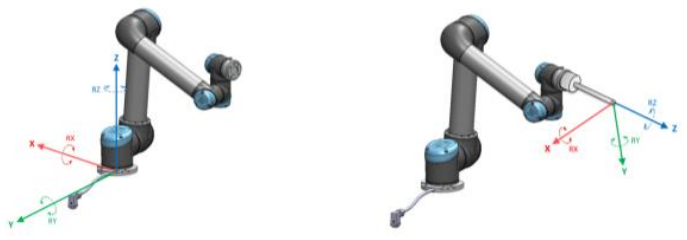


Figure 2.1.1: Base frame (left) and the TCP frame (right) [85].

The positive y-axis is defined along the power chord of the robot, the positive z-axis is defined upwards, and the positive x-axis can be derived using the right hand rule.

Kinematics allow conversions between these two spaces in what is known as forward and inverse kinematics. Forward kinematics is the method of calculating the unique pose of a given frame through given joint angles. Inverse kinematics is the calculation to get the joint angles for a given pose and does not guarantee that the solution exists or is unique. Figure 2.1.2 shows the relationship between both spaces, where the n corresponds to the number of joints in the manipulator, eg. for this application, the joint space will be represented by $(q_1, q_2, q_3, q_4, q_5, q_6)^T$:

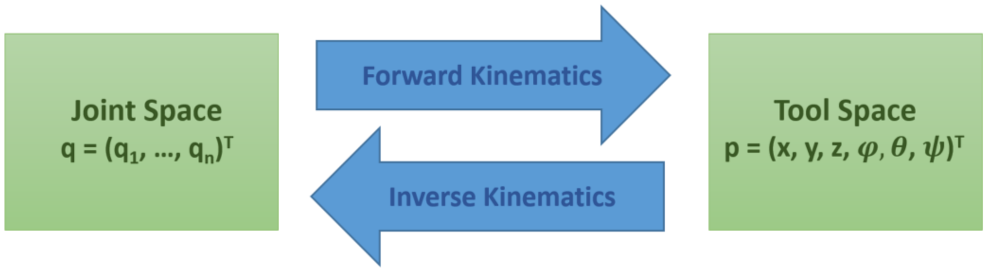
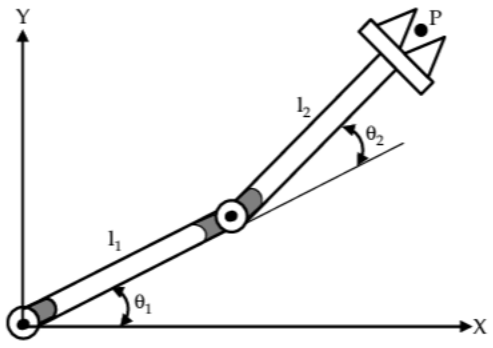

Joint Space
$q = (q_1, \ldots, q_n)^T$
Forward Kinematics
Inverse Kinematics
Tool Space
$p = (x, y, z, \varphi, \theta, \psi)^T$

Figure 2.1.2: Converting between joint- and tool spaces using forward- and inverse kinematics [19].

The following simplified example will give an idea of how forward and inverse kinematics are related:



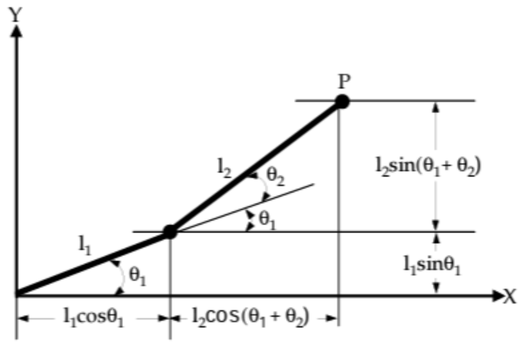


Figure 2.1.3: Simple example for forward- and inverse kinematics using two links [32].

Using geometry, the 2D point P can be separated into its x- and y- components as follows:

$P_x = l_1\cos(\theta_1) + l_2\cos(\theta_1 + \theta_2)$

$P_y = l_1\text{Sin}(\theta_1) + l_2\ \text{Sin}(\theta_1 + \theta_2)$

For forward kinematics, P_x and P_y are unknowns that need to be solved for and the θ_1 and θ_2 are given, which leads to one unique solution by plugging in the thetas into the formulas. For inverse kinematics, however, θ_1 and θ_2 are then the unknowns and the P_x and P_y are given. The solution can then be found using numerical methods, but multiple values may be the solution or there could be no values that solve the problem.

For Fluxana's pick and place applications both types of movements are used. For example, to go to the home position, the joint space needs to be used since at this configuration, the manipulator arm is fully stretched which causes problems with finding the inverse kinematics solutions due to singularities or loss of degrees of freedom. However, linear movement is needed when going up or down to load or unload the crucible or the glass bead to make sure that the end effector does not collide with the components during movement. However, the motion planning and the choice of space to execute the motion in will be done using MoveIt! Framework for the scope of this study. For more information, please refer to the MoveIt! documentation in the ROS chapter. To use tool space, it is important to learn about the different ways of representing the orientation of the TCP, which will be explained next.

2.1.1. ORIENTATION REPRESENTATION

Rigid bodies are assumed to not deform during movement, which is an idealization that will be used in this study. Rigid body motion can be:

- Translation: whole body moves at the same velocity and acceleration
- Rotation: movement around an axis of rotation.
- Planar motion: combination of translations and rotations

To move the robot, the target joint values or pose need to be known. The term pose describes the position and orientation of a frame of reference. The position is a vector from the origin of the base frame to the origin of the reference frame of the rigid body. While the orientation describes the rotation of the frame of reference in space, and can be represented using a rotation matrix, Euler angles, rotation vector, or quaternions as explained in more detail in this section.

To understand more what a frame is, here are some examples of standard frames in a workspace:

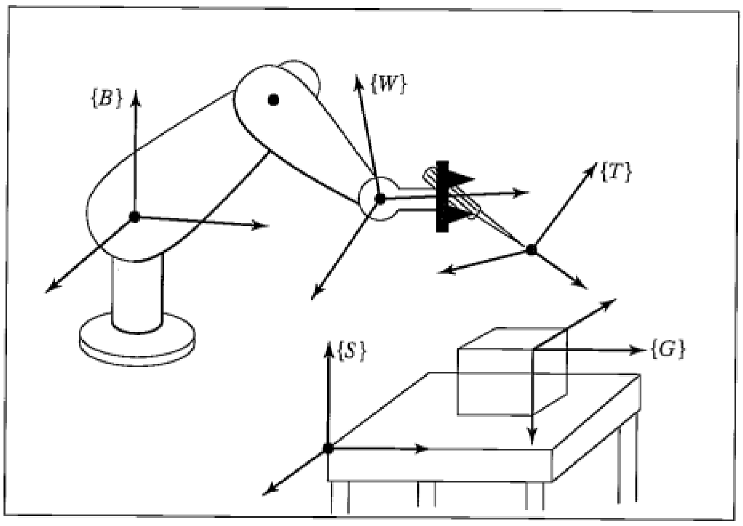


Figure 2.1.4: Standard frames examples [15].

Where:

- Base frame {B}: the frame of the base link.
- Station frame {S}: the "world" frame, where the robot actions are performed relative to it, here it is given as the table corner.
- Wrist frame {W}: the frame of the last manipulator link, in case of UR3e it is the wrist 3.
- Tool frame {T}: the frame attached to the tool of the manipulator, in this study, the gripper.
- Goal Frame {G}: the frame of where the robot is to move its tool frame.

2.1.1.1. ROTATION MATRIX

The rotation matrix is a $3x3$ matrix describing the relative orientation between reference frames. The rotation matrix describing $\{A\}$ relative to $\{B\}$ is denoted as:

$$ {}_{B}^{A}R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} $$

Where $\{A\}$ and $\{B\}$ are the short-hand representations of frames A and B. Multiple rotation matrices can be multiplied to represent multiple rotations. Since matrices are not commutative, the order of rotations matters.

The rotation that makes up the rotation matrix can be separated into their rotation components around the x-, y-, and z-axes as follows:

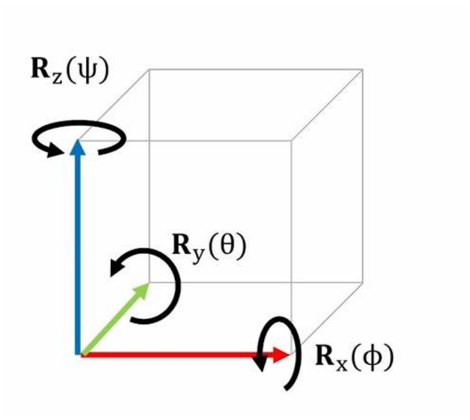


Figure 2.1.5: Visual representation of the rotation matrix [99].

2.1.1.2. EULER ANGLES

Euler angles describes the orientation of a rigid body in reference to a fixed frame and uses the roll, pitch, and yaw (RPY) to represent the orientation, where:

- Roll is angle phi (φ) about the x-axis.
- Pitch is the angle theta (θ) about the y-axis.
- Yaw is the angle psi (ψ) about the z-axis.

The following is a visual representation of how the Euler angles can change the paper plane orientation in space:

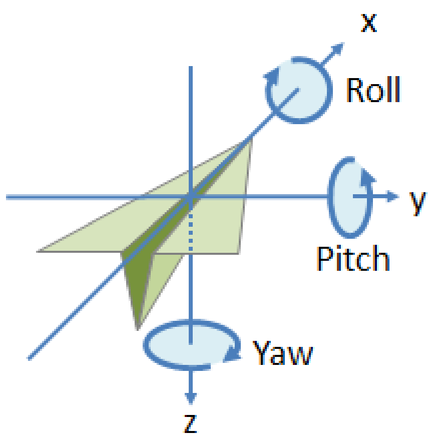


Figure 2.1.6: Visual aid for the roll pitch and yaw [6].

There are 12 possible combinations for Euler angle representations, but the UR3e uses R_{zyx}. Euler angles should be converted into the rotation matrices for kinematics and dynamics calculations using the following rotation matrix representations of RPY:

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi) & \cos(\varphi) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying the rotation matrices in the order used by Universal Robots yields the rotation matrix:

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma)$$

$$= \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

However, Euler angles have the disadvantage of containing discontinuities and that the result depends on the order of rotations. Therefore, Universal Robots uses the rotation vector representation. URScript provides functions that convert between the two representations and are *rpy2rotvec* and *rotvec2rpy*, for more information on how to use those functions, please refer to the URScript manual.

An important point to note about the RPY angles is the possibility of the occurrence of the gimbal lock [99], which is defined by the loss of a degree of freedom and occurs when the pitch angle is $\frac{pi}{2}$ radians, this effect can be illustrated by Figure 2.1.7.

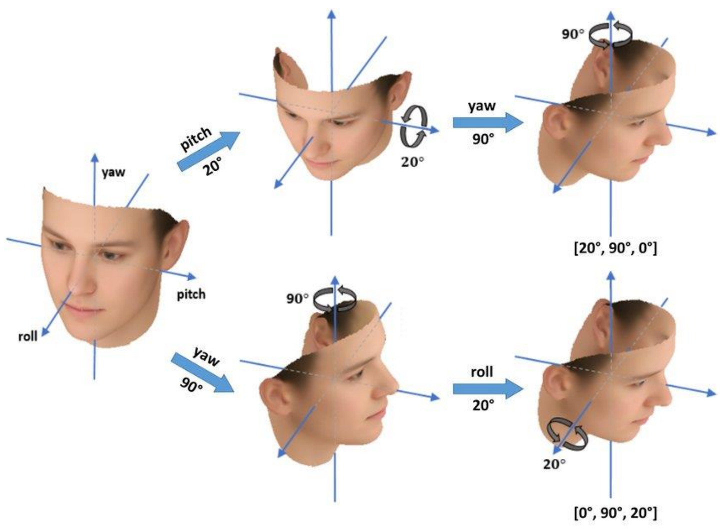


Figure 2.1.7: Both [20°, 90°, 0°] and [0°, 90°, 20°] lead to the same result due to the gimbal lock [37].

To avoid this problem, ROS uses quaternion representation of orientation. In brief, conversions between Euler and rotation vector are necessary when communicating with the teach pendant and conversions between Euler and quaternions are necessary when communicating with ROS.

2.1.1.3. ROTATION VECTOR

The rotation vector representation uses a unit vector to define the rotation axis and an angle θ for the magnitude of rotation about the axis and is used for the default pose display on the Universal Robots teach pendant, where it is defined as follows:

$$r = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} \theta u_x \\ \theta u_y \\ \theta u_z \end{bmatrix}$$

The following is a visual aid for the rotation vector representation:

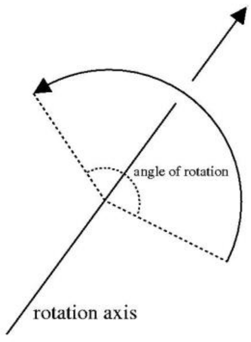


Figure 2.1.8: Rotation vector visual representation [6].

2.1.1.4. QUATERNIONS

Quaternions use four values to fully define the rotation of a vector in comparison to the nine values from a rotation matrix. They are based on complex numbers and are defined as [27]:

$$\boldsymbol{q} = \boldsymbol{w} + \boldsymbol{x}\,\mathbb{i} + \boldsymbol{y}\,\mathbb{j} + \boldsymbol{z}\,\mathbb{k},\ \{\boldsymbol{w},\ \boldsymbol{x},\ \boldsymbol{y},\ \boldsymbol{z} \in \mathbb{R}\}$$

Where:

$$\mathbb{i}^2 = \mathbb{j}^2 = \mathbb{k}^2 = \mathbb{i}\mathbb{j}\mathbb{k} = -\mathbf{1}$$

Quaternions do not have the problem of gimbal lock and allow for seamless interpolation between three dimensional orientations as well as avoid numerical errors that arise from using other methods. Given a point p, the rotated version is calculated by [2]:

$$p_{rot} = q * p * q^{-1}$$

Where:

$$q^{-1} = w - x\,\mathbb{i} - y\,\mathbb{j} - z\,\mathbb{k}$$

The conversion between Euler angles and quaternions are done through the $tf.transformations.quaternion\_from\_euler$ and $tf.transformations.euler\_from\_quaternion$ functions. For more information, please refer to the ros.org documentation of the function, which can be found in the following referenced source [97].

2.1.2. KINEMATIC CHAIN

Kinematic chain describes how the manipulator arm elements are arranged and connected. These elements can be links, joints, or end-effectors. The UR3e is made up of a stationary base link to connect the robot to the workstation, as well as moveable links that are connected using revolute joints, in addition to a Robotiq HandE end-effector that is used for gripping. The mentioned elements are briefly described below:

- Base: a stationary link used to attach the robot arm to the table, since the manipulator arm is fixed as most industrial robots are.
- Link: rigid part that has a defined relationship with the other links. The UR3e has six movable links.
- Joint: elements that connect links in a kinematic chain. There are two primary types of joints:
    - Prismatic: translates a frame along an axis.
    - Revolute: rotates a frame around an axis.

The UR3e has six revolute joints which provide six degrees of freedoms (DOFs) in total and allow for relative rotational motion between links, where DOFs define the ways the manipulator arm can move. The arrangement of joints in the UR3e can be seen as follows:

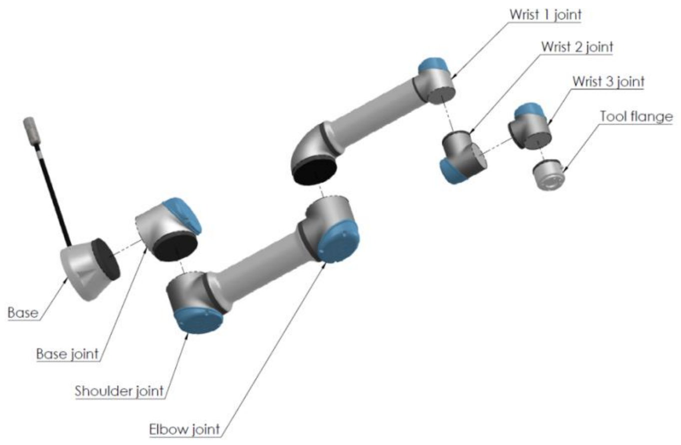


Figure 2.1.9: Breakdown of the UR3e Joints [85]

- End-effector: A Robotiq HandE gripper that is used for the pick and place application. The HandE gripper is a 1kg end effector with gripping force of 20-185N and gripping speed 20-150mm/s based on user defined settings and can carry a load of up to 5kg. Controlling the gripper is outside the scope of this study, however, it will be considered in the collision checking when giving the robot move commands, this is done by adding the digital description of the gripper to the URDF file of the manipulator, which will be discussed in more detail in the URDF section in the ROS documentation. The TCP is defined in relation to the base reference system and can be used in calculating paths for the end effector.

The following image shows an example of the different kinematic chain elements of the UR3e:

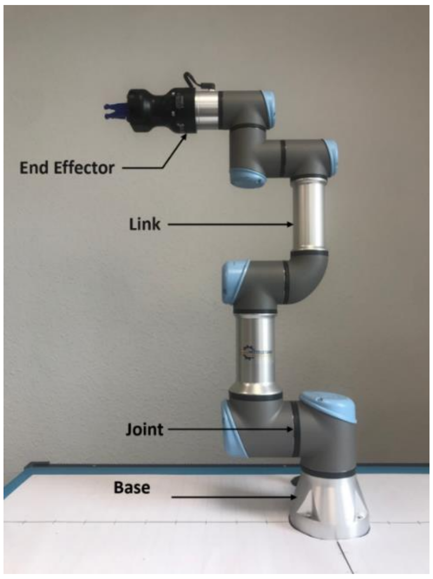


Figure 2.1.10: Kinematic chain elements of the UR3e.

Figure 2.1.11 shows the coordinate system used for the same pose the robot has in the previous figure. The figure is taken from the digital representation of the robot by using RViz which will be explained in more details in the RViz section ROS documentation. The green line represents the positive y-axis, the red line represents the positive x-axis, and the blue line represents the positive z-axis:

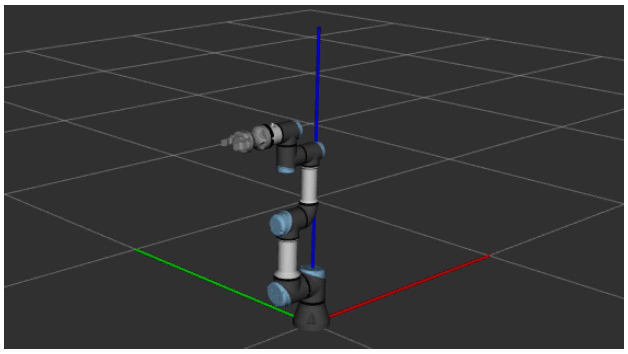


Figure 2.1.11: Coordinate system used by the robot: x-axis in red, y-axis in green, z-axis in blue.

Now that the different kinematic chain elements have been described, it is important to be able to describe the relationship between them, which can be done using the DH parameters.

2.1.3. DH PARAMETERS

The relationship between the different links and joints of the manipulator arm needs to be defined to do forward and inverse kinematics calculations. For the definition of the kinematic chain, the four Denavit-Hartentberg (DH) parameters need to be known. These parameters are defined from the base until the end effector and are used to define the link frames. DH parameters consist of two rotational parameters and two linear ones and are as follows:

- d_i: distance between the origin of frame$_{n-1}$ and frame$_n$, along the direction Z_{n-1}.
- θ_i: rotation about axis Z_{n-1} for axes X_{n-1} and X_n to align.
- r_i: distance between the origin of frame$_{n-1}$ and frame$_n$ along the X_n direction.
- α_i: rotation around axis X_n for axes Z_{n-1} and Z_n to align.

The following figure is used to define the DH parameters of the UR3e when it has joint value zero for all joints.

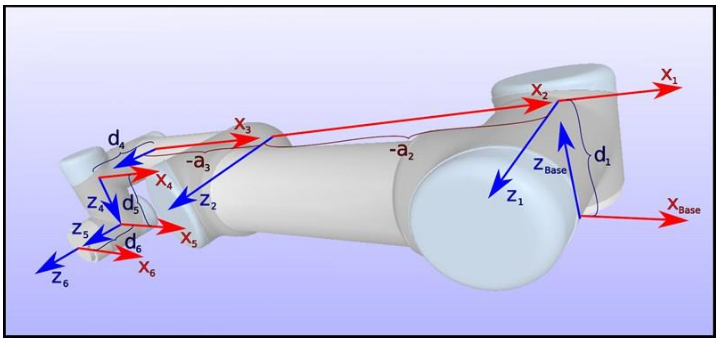


Figure 2.1.12: The different link frames for a UR3e at the joint values [0, 0, 0, 0, 0, 0] [87].

α_i and r_i are constants that are derived from the geometry of the kinematic chain and θ_i and d_i depend on whether the joint is revolute or prismatic. For the UR3e, the joints are revolute, therefore, d_i is constant and θ_i is variable [71]. The UR3e DH Parameters are defined as follows:

	theta [rad]	a [mm]	d [mm]	alpha [rad]
Joint 1	0	0	151.85	$\frac{\pi}{2}$
Joint 2	0	-243.55	0	0
Joint 3	0	-213.2	0	0
Joint 4	0	0	131.05	$\frac{\pi}{2}$
Joint 5	0	0	853.5	$-\frac{\pi}{2}$
Joint 6	0	0	921	0

Table 2.1.1: The DH Parameters for UR3e [87]

These parameters will play an important role in defining the kinematic chain digitally for ROS using the URDF and Xacro files in later chapters.

2.1.4. TRANSFORMATIONS

A transformation matrix combines rotation and translation between frames in a $4x4$ matrix, where ${}^{A}_{B}T$ is the transform matrix of {B} relative to {A} and has the following formula:

$$ {}^{A}_{B}T = \begin{bmatrix} {}^{A}_{B}R & {}^{A}P_{BORG} \\ 0\ 0\ 0 & 1 \end{bmatrix} $$

Figure 2.1.13 shows the transform of {B} relative to {A}, where the origin of {B} is offset from the origin of {A} by ${}^{A}P_{BORG}$. Frame {B} is also rotated with respect to {A} with what can be described using the rotation matrix:

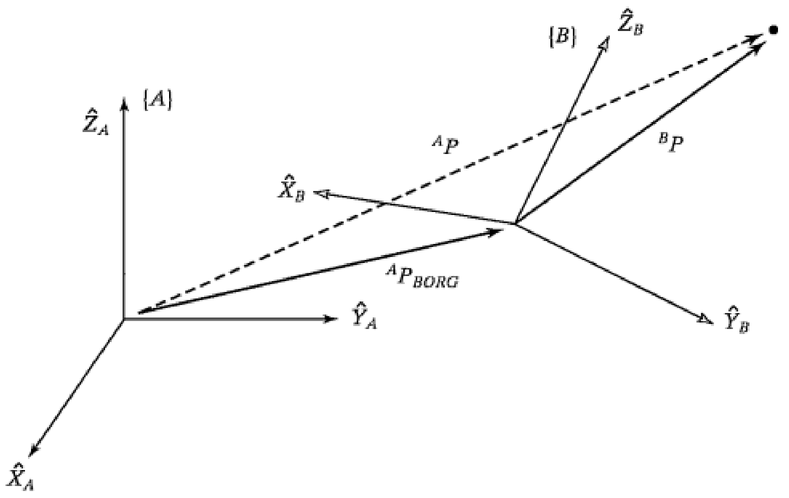


Figure 2.1.13: Transform of {B} relative to {A} [15].

To describe relative relationships between links, {i} is assigned to link i. The link transformations between links i relative to $i - 1$ can be denoted as ${}^{i-1}_{i}T$, where c and s are the shorthand notations for cosine and sine respectively and parameters used are DH parameters:

$$ {}^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Cascading can be used to get position and orientation of frames not directly neighboring each other. Transformation of {n} relative to base frame is defined as:

$$ {}^{0}_{n}T = {}^{0}_{1}T\,{}^{1}_{2}T \ldots {}^{n-1}_{n}T $$

If the transform of {B} relative to {A} is known, the transform of {A} relative to {B} is the inverse as by:

$$ {}^{A}_{B}T = {}^{A}_{B}T^{-1} $$

This is especially helpful when the transformation of {A} relative to {B} is known but the transformation of {B} relative to {A} is the one needed for a calculation. To calculate the coordinates of a point in {B} relative to {A}, the following formula is used:

$$ {}^{A}P = {}^{A}_{B}T\,{}^{B}P $$

Some of these formulas will be used later to calibrate the TCP on the teach pendant to get matching results with ROS.

2.1.5. JACOBIAN MATRIX

The Jacobian matrix is a matrix containing the velocity vectors by taking the partial derivatives of the joint and link positions. It relates the velocity of the end effector and the joints through the following equation:

$$ \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = J \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \\ \dot{q}_6 \end{bmatrix} $$

Where q_i is the velocity of the ith joint, and the J is a 6x6 Jacobian matrix since the UR3e has 6 joints. This matrix can be used in the future to take the input for the speed as a $1x6$ vector specifying the speed in tool space and calculating the corresponding joint speeds and setting them accordingly, but that exceeds the scope of this study.

2.2. ROBOT OPERATING SYSTEM (ROS)

ROS is an open-source middleware that offers libraries and tools for scalable robot applications [16]. These applications include:

- Robot control: sending move commands that have been checked for possible collisions with the environment is done with path planning using the MoveIt! framework.
- Visualization: is useful for getting the perception of the robot of its world and objects in its environment, and is, therefore, a helpful debugging tool. RViz has proven very useful when getting the robot's digital description of the kinematic chain and frames to match the real robot during the course of this study.
- Simulation: uses a digital twin of the robot and can be used for testing and analysis of the robot behaviors before executing commands on the real robot, thereby enhancing safety. The tool for simulating in ROS is Gazebo but it exceeds the scope of this study.
- Grasping: grasping and releasing gripper actions are essential in the given pick and place application as crucibles and glass beads need to be grasped and placed at the appropriate positions. The Fluxana Library does not include gripping functions for the scope of this study, but it will be added on in the future.

ROS filesystem has the basic structure outlined below in Figure 2.2.1, where a Catkin workspace contains folders for source-, build-, and development related software. ROS packages can be found in the src folder and contain files with information about dependencies of the packages and build information, this information is found in the CMakeLists.txt and the package.xml files. The scripts folder in the package contains executables like the .py files containing the Fluxana Library code in the case of this study. The package can also include folders under the name srv or msg with ROS messages that are custom created for the package, ROS messages are used for defining different fields and constants and has different structures based on the different types of communication in ROS and will be discussed later in the ROS Messages section. The launch folder in the package contains launch files that can be used to run one or more nodes as well as define required parameters. Since the Workspace created for this study does not need all the folders mentioned in the image below, the actual project will have a different structure that will be discussed later in the system architecture.

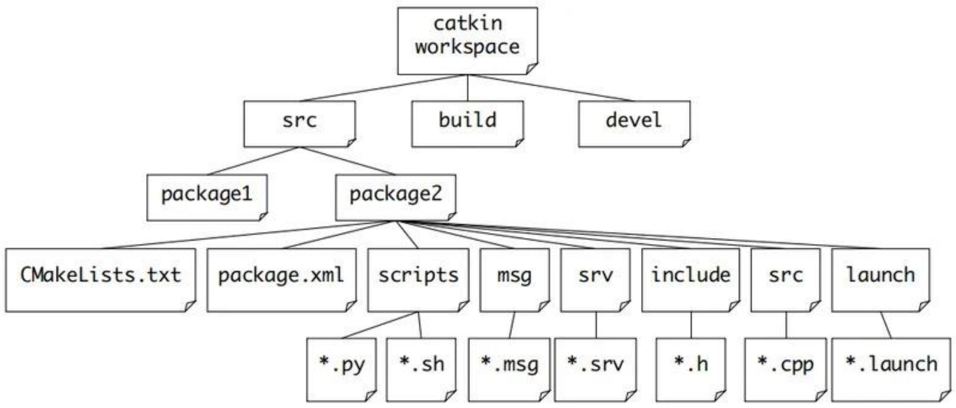


Figure 2.2.1: File structure for a ROS workspace [4].

ROS uses nodes to perform computations, where different styles of communication can be used and are as follows:

- Topics: use the publish/subscribe communication paradigm and allows for asynchronous communication between nodes. A publisher publishes information to a topic and subscriber to that topic receives it.
- Services: use the request/response paradigm, where a client sends a request with the required arguments and receives a response from the server. This is done in a synchronous manner, meaning that the client does not execute anything else until it receives the response.
- Actions: uses similar communication to services but does not block the client until the response is received. Actions are useful for tasks that execute over a long duration and require feedback or the option for cancellation.

To use ROS, a ROS master needs to be running as described below in more detail.

2.2.1. ROS MASTER

ROS master acts as a tracker that lets publishers, subscribers and services locate each other by providing naming registration services to the nodes to allow the nodes to communicate on a peer-to-peer level [100]. A publishing node first needs to advertise its data but if there are no subscribers to the topic, no data is sent. When a node subscribes to the publishing node a connection is established between them and the data transfer starts [102]. Figure 2.2.2 shows how the Master works:

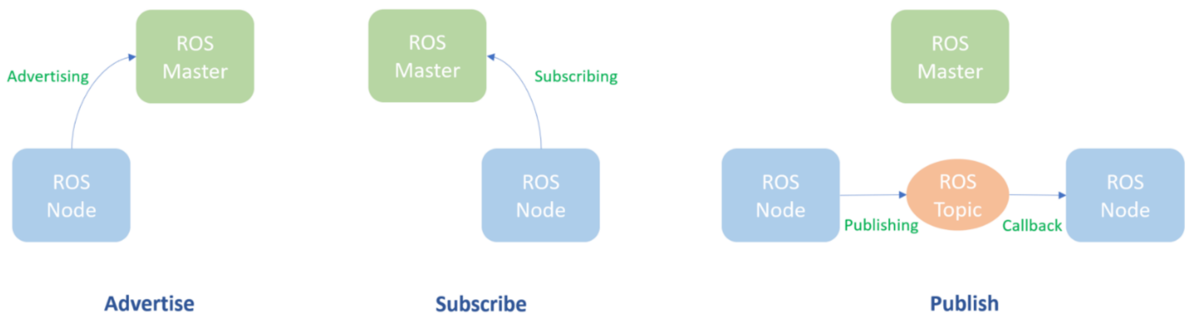


Figure 2.2.2: The steps for publishing and subscribing in ROS [63] [64].

ROS transports message using TCP/IP-based protocol called TCPROS or a UDP-based one called UDPROS, where TCPROS is the default protocol for message transfer due to the lossy nature of UDPROS. TCP/IP or Transmission Control Protocol/Internet Protocol defines how devices transfer data to each other over a network. TCP/IP involves breaking down the data to be transferred into packets that are sent and reassembled at the receiver. The TCP part defines the rules for data transfer and the IP is for defining the address the data is transferred to.

This protocol has four layers as shown below:

Application layer: the level on which the user communication is done.

Transport layer: responsible for data transfer between two devices and acknowledgement of successfully received data packets.

Network layer: controls the movement of data packets within the network.

Datalink layer: responsible for sending and receiving data using an Ethernet cable, wireless network, ...

Figure 2.2.3: The different layers of the TCP/IP protocol[96][20].

Running a ROS system on multiple machines needs only one master, with all nodes configured to that master [25]. This setup will be used to communicate between the GUI for the Fluxana Library and the Linux machine in the future and will be discussed further in the future Works section. Next the file structure of ROS will be described in more detail, starting with the ROS Workspace.

2.2.2. ROS WORKSPACE

A ROS workspace is where ROS packages are organized and built, ROS packages will be discussed in more detail in the next section. ROS melodic uses catkin build system to generate libraries, executable programs, and generated scripts. To create a Catkin workspace, use the following commands in Linux [40]:

```
1.  $ mkdir -p ~/catkin_ws/src
2.  $ cd ~/catkin_ws/
3.  $ catkin_make
4.  $ source devel/setup.bash
```

Explanation:

Line 1 creates a folder called catkin_ws and within that folder one called src. Line 2 then continues to navigate inside the catkin_ws folder to build it and source it to update the changes in the environment in the lines to follow. The information necessary to build software packages can be found in CMakeLists.txt [91]. Dependency management is done in the package.xml file. ROS Packages can then be placed inside the src folder.

2.2.3. ROS PACKAGE

ROS packages are directories used for organizing software, configuration, launch parameter files, and nodes for easy code reuse. To create a package, use the command, all letters in the package name must be lower case to avoid issuing errors:

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

The [dependX] specifies the dependencies of the package. A ROS package commonly contain the following folders:

- scripts: contains executables
- src: contains source files
- launch folder: contains launch files
- package.xml file: an xml file used for storing information about the package dependencies. Different types of dependencies can be present in the package.xml file, and can be seen in Table2.2.1 [8]:

Type	Tag	Description
Build dependencies	<build_depend>	specifies the additional packages needed at build time to build the package
Build export dependencies	<build_export_depend>	Needed additional packages needed for building libraries against the packages
Build tool dependencies	<buildtool_depend>	specifies build tool needed, in this case catkin

Table 2.2.1: Excerpt of package.xml dependencies

- CMakeLists.txt file: used for storing information for building a package, where the ordering of the file elements is predefined and should be followed. The most relevant tags to the thesis are listed in Table 2.2.2 [70]:

Command	Description
cmake_minimum_required()	CMake Version
project()	Package name
find_package()	Packages needed for the build
add_message_files(), add_service_files(), add_action_files()	Message, service, or action generators
generate_messages()	Invoke message, service, or action generation

Table 2.2.2: Excerpt of CMakeLists.txt tags

To navigate to the location of a package from the terminal, use:

```
roscd <package_name>
```

Several packages can be grouped together into a stack, which will be explained next. After that, nodes will be explained. Nodes are executables in a package that are used to do computations.

2.2.4. ROS STACK

ROS Packages that serve common purpose can be further grouped together to form a ROS stack for easier code sharing. An example of ROS Stack is a control stacks which contains packages that provide the necessary functionalities to control the robot. This includes motion planning, joint control, and getting sensor information for feedback. The command-line tool rosstack is used for getting relevant information about ROS Stacks, these commands could be to list the available ROS stacks in a file system or to calculate the dependency tree of the stack [23][98].

2.2.5. ROS NODE

A node is an executable that performs computations and can communicate with other nodes through publishing or subscribing or providing a service [63]. A node can be written mainly in python or C++, but other languages can used to write a ROS node like Lisp and Java and more. ROS is also language agnostic, meaning that multiple nodes can be written in different languages and still communicate with each other. Each running node should have a unique identifier. Therefore, it is important not initialize multiple nodes with the same name to avoid the deletion of one of the created nodes. This unique naming can be done by setting the anonymous parameter to True to ensure that the node has a unique name as demonstrated below [26]:

```
rospy.init_node("<node_name>", anonymous = True)
```

The command-line tool for displaying information about ROS nodes is rosnode and supports commands like [66]:

Command	Description
rosnode info <node_name>	displays node information like publishers, subscribers, and type
rosnode kill < node_name>	kills one or more running nodes by name
rosnode list	displays the active nodes

Table 2.2.3: Excerpt of rosnode command line tool commands.

Nodes can communicate with each other through topics, services, and actions. Single or multiple nodes can be run using a launch file. Each of these concepts will be described in more detail in the following sections.

2.2.6. LAUNCH FILE

Launch files are XML files with the .launch extension. A launch file can be used to launch multiple nodes at once as well as configure their parameters. The following tags can be used in a launch file[69]:

- <launch>: root tag which encloses all the other tags
- <node>: specifies the node to be run
- <group>: groups multiple nodes
- <arg>: declares an argument
- <include>: includes other launch files
- <rosparam>: sets a parameter using a rosparam file
- <param>: sets a parameter using the parameter server

To execute a launch file, the following command outline is used:

```
roslaunch <package_name> <launch_file>
```

To launch ROS, MoveIt!, RViz for this study, the following commands are used, where the MoveIt! framework and RViz will be described in more details in their respective sections. The command to start the Universal Robots ROS driver requires two arguments, one of which is the robot's IP address of 192.168.11.100 as can be seen below:

To start the Universal Robot ROS driver

```
roslaunch ur_robot_driver ur3e_bringup.launch robot_ip:=192.168.11.100
use_tool_communication:=true
```

To start MoveIt!

```
roslaunch ur_hande_moveit_config ur3e_moveit_planning_execution.launch
```

To start Rviz

```
roslaunch ur_hande_moveit_config moveit_rviz.launch
```

2.2.7. ROS TOPIC

Topics are channels for unidirectional message exchange between nodes. Nodes that want to retrieve data subscribe to a topic, while nodes that generate data publish to a topic [19]. Multiple nodes can publish/subscribe to the same topic and a node can subscribe/publish to multiple topics. The type of the topic is the ROS message it uses. This type needs to be adhered to when publishing data.

The command-line tool rostopic can be used to display information about ROS topics and supports commands such as [93]:

Command	Description
rostopic echo <topic_name>	Displays the messages published to a topic
rostopic echo <topic_name/field>	Display a specific field in a message
rostopic info <topic_name>	Displays information about the topic
rostopic list	Displays the current topics
rostopic pub <topic_name> <topic_type> <data>	Publishes data to a topic

Table 2.2.4: Excerpt of rostopic command line tool commands.

For information about writing a publisher and subscriber in python, please refer to the ROS Wiki documentation for "Writing a Simple Publisher and Subscriber (Python)".

2.2.8. ROS SERVICE

Services are a synchronous method of communication that use a server and a client for message transfer because the unidirectional way of communication of publisher/subscriber is not always suitable for the request/reply paradigm. When a service is called, the program waits for the response and does not execute anything in that time. A simple illustration of how services work is as follows:

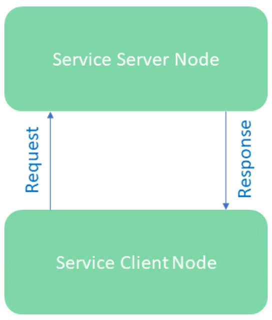


Figure 2.2.4: Flow of communication between the ROS Service server and client [29].

The command line tool rosservice supports helpful commands such as [14]:

Command	Description
rosservice call <service_name> [service_args]	Calls a service provided the arguments
rosservice info <service_name>	Displays information about a service
rosservice list	Displays a list of all services

Table 2.2.5: Excerpt of rosservice command line tool commands.

For more information about writing a service server and client in python please refer to the ROS Wiki for "Writing a Simple Service and Client (Python)".

2.2.9. ROS ACTION

An action is asynchronous client-to-server communication that can be thought of as creating a thread to perform a command alongside main tasks as the system waits for the response. When a goal is sent for execution, feedback messages of the current progress of the execution are sent out until the goal is executed. At the end of the execution, a result message is sent out. A goal can be cancelled or replaced by a new goal during execution. This information can be seen in Figure 2.2.5:

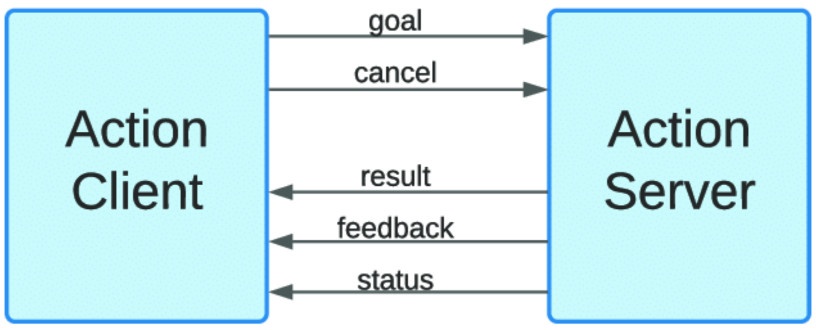


Figure 2.2.5: Shows the structure of a ROS Action and the flow of communication between the server and client[44].

For more information about writing an action server and client in python please refer to the ROS Wiki documentation.

2.2.10. ROS MESSAGES

Messages used by topics, services, and actions have different structures due to the different nature of each method of communication. The respective extensions are shown in the figure below:

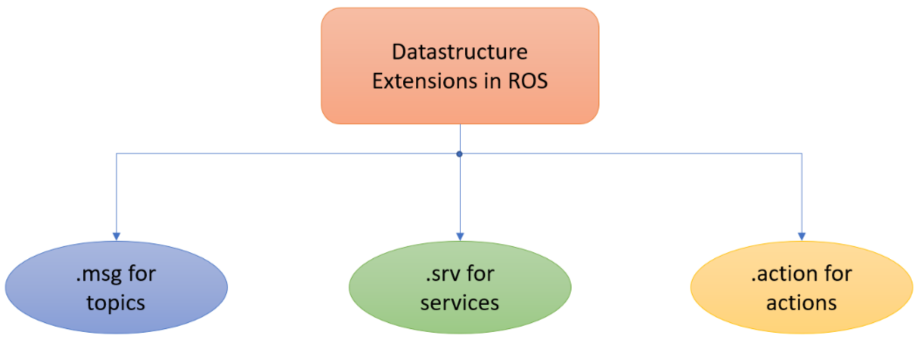


Figure 2.2.6: Different message types and extensions of messages used in ROS.

These messages can have fields and constants of a primitive or a custom datatype. To declare data fields, the following format is used:

```
field_type1 field_name1
```

While constants are declared as follows:

```
Constant_type1 CONSTANT_NAME1 = constant_value1
```

Each of the message types will now be discussed in more detail.

2.2.10.1. ROS TOPIC MESSAGES

Topics use a .msg message for their communication and is defined simply as a list of fields and constant [9]. Below is a simple example of a ROS message which has an integer constant, that the name X and value 123, it also has a field named Y, and a string field name FOO:

```
int32 X=123
int32 Y
string FOO
```

The command-line tool rosmsg displays information about ROS messages and supports commands such as:

Command	Description
rosmsg info <message_type >	Displays the fields of the ROS message
rosmsg list	Displays a list of all messages

Table 2.2.6: Excerpt of rosmsg command line tool commands.

2.2.10.2. ROS SERVICE MESSAGES

This message type enables the request/response communication. Request is what is taken as an argument for the service, and response is what is returned by the service. The message definition divides each of the request and response definitions with three dashes [67]. Below is a simple example of a service message which takes a string named str_request as an argument and returns a string named str_response:

```
string str_request
---
string str_response
```

The rossrv command line tool is used to display information services and contains functions like:

Command	Description
rossrv info <service_message_type >	Displays the fields of the ROS service message
rossrv list	Displays a list of all service messages

Table 2.2.7: Excerpt of rossrv command line tool commands.

2.2.10.3. ROS ACTION MESSAGES

These message types are used by action servers and action clients. The action message specifications are goal, result, and feedback and are defined in a file with the extension .action with each specification definition separated from the next by three dashes, where:

- Goal: a task that is sent by the action client to the action server to be executed.
- Result: upon completion of the goal execution, the result is sent to the action client.
- Feedback: contains information about the progress of the goal execution

The following is an example of an action message with an int32 goal named order, an int32 array result called sequence and an int32 array feedback called sequence [35].

```
#goal definition
int32 order
---
#result definition
int32[] sequence
---
#feedback
int32[] sequence
```

Now that the basic concepts of ROS are explained other sections will explain how motion planning is done with MoveIt! Framework and visualization is done with RViz using the digital description of the robot in the form of URDF and Xacro files.

2.3. MOVEIT! FRAMEWORK

MoveIt! is a framework that does calculation necessary for motion planning, collision checking, grasp generation, as well as kinematic analysis [50]. Visualizing the planning and execution processes can be done with RViz or simulated using Gazebo. The following image describes the architecture of the MoveIt framework, where the relevant concepts to this study like the Move Group node, the MoveIt! Commander python interface along with the inverse kinematics solver and some planner libraries such as OMPL, CHOMP, and STOMP will also be explained.

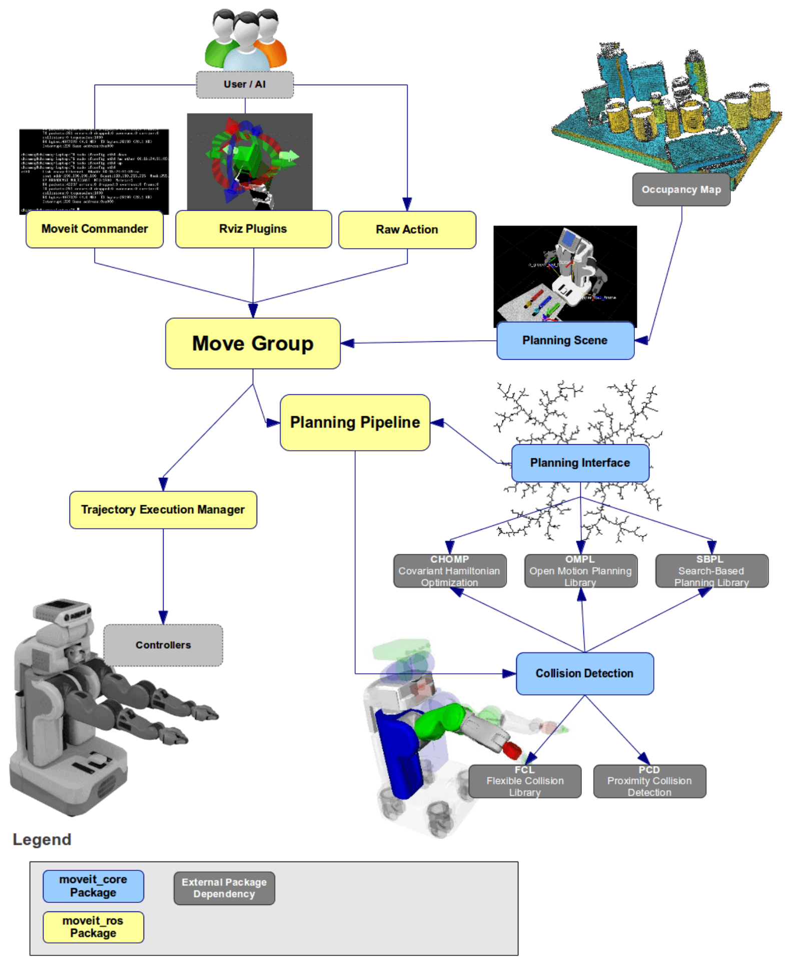


Figure 2.3.1: Architecture of MoveIt! Framework [12].

The move-group primary node brings together the different components of MoveIt!. The move_group calls services and actions depending on the user application as well as collects information about transforms and the robot state. The robot representation, kinematic chain definition, and constraints for the MoveIt! package are given as URDF, SRDF, and config files, where:

- Unified Robot Description Format URDF (URDF) is a digital model of the robot and will be discussed in more detail in its own section.
- Semantic Robot Description Format (SRDF): can be created using the Moveit! Setup assistant and is used to specify the following information:
    - Virtual joints: creates a fixed joint that connects the manipulator arm to the world
    - Passive joints: joints that are not actuated in the robot
    - Groups: used to group together joints or links that have a common purpose. For this application, there is a group for the manipulator arm and one for the gripper.
    - Kinematic Chains: a group of links that can be grouped together using joints to represent an entity.
    - End-effectors: used to add the gripper to the manipulator arm
    - Self-collisions: used for disabling the collision checking between links that are always in collision or never in collision to save up on the processing time

  - Robot poses: Used to predefine positions that can be used in planning, for example the joint values for the home position
- config: A set of files that define the joint limits, as well as visual and physical parameters. These files are automatically generated by MoveIt! setup assistant but can be changed to optimize the parameters according to the application.

In this study, MoveIt! functions will be used in python through the Moveit! commander which will now be explained in more detail.

2.3.1. MOVEIT! COMMANDER PYTHON INTERFACE

The MoveIt! Commander python interface is a wrapper that includes functions to set target pose, or joint values, create and execute motion plans in the MoveGroupCommander class, as well as attach and detach objects from the robot for collision checking in the PlanningSceneInterface class [48], and get information about the planning frame, joints, and links with the RobotCommander class. To use functions from the previously mentioned libraries in Fluxana Library, an instance of the MoveIt! commander as well as instances of each class needs to be initialized. To initialize the moveit_commander the following command is used:

```
moveit_commander.roscpp_initialize(sys.argv)
```

This wrapper is divided into three parts, which are the RobotCommander, PlanningSceneInterface, and MoveGroupCommander:

- MoveGroupCommander: is a library whose functions are used for motion planning and executing move commands and getting the robot’s current joint values or pose as well as functions for a pick and place application. To instantiate a MoveCommanderGroup object to be able to plan and execute move commands, use the following command:

```
group_name = “arm”
group = moveit_commander.MoveGroupCommander(group_name)
```

- PlanningSceneInterface: is a library with functions that add an object to be taken into consideration in the collision calculations, as well as get information about those attached objects [74]. To instantiate a Planning_Scene_Interface object to be able to update the state of the robot with respect to its “world” use the following command:

```
scene = moveit_commander.PlanningSceneInterface()
```

- RobotCommander: is a library with functions related to getting information about joints and links like their names as well as information about the frame in reference to which the planning is done [73]. To instantiate a Robot_Commander object to provide kinematics information about the robot use the following command:

```
robot = moveit_commander.RobotCommander()
```

2.3.2. MOTION PLANNING

An important concept when learning about MoveIt! is motion planning. Motion planning is considering the robot's constraints when going from a start pose to an end pose without collisions. Prior to planning a read only lock is set on the planning scene to avoid changes happening to it while planning [46]. To do motion planning, Open Motion Planning Library (OMPL) is used in this study, which is a wrapper for Flexible Collision Library (FCL) and Proximity Query Package (PQP), which are libraries for collision detection, distance computation, and tolerance verification [47][51][52]. When moving between targets, OMPL divides the path into a series of discrete times for collision checking rather than continuous collision checking to save up resources. Many OMPL planners, seek having a higher speed of finding the solution path over the quality of the path. This path is then smoothened and processed to get it closer to optimal. The planner used by OMPL in this thesis is Rapidly Exploring Random Tree (RRT) which is an algorithm for path planning problems that aims to incrementally decrease the distance between the tree branches it creates and a chosen point. RRT is used for solving inverse kinematics problems and collision free control. Figure 2.3.2 visually represents how RRT works:

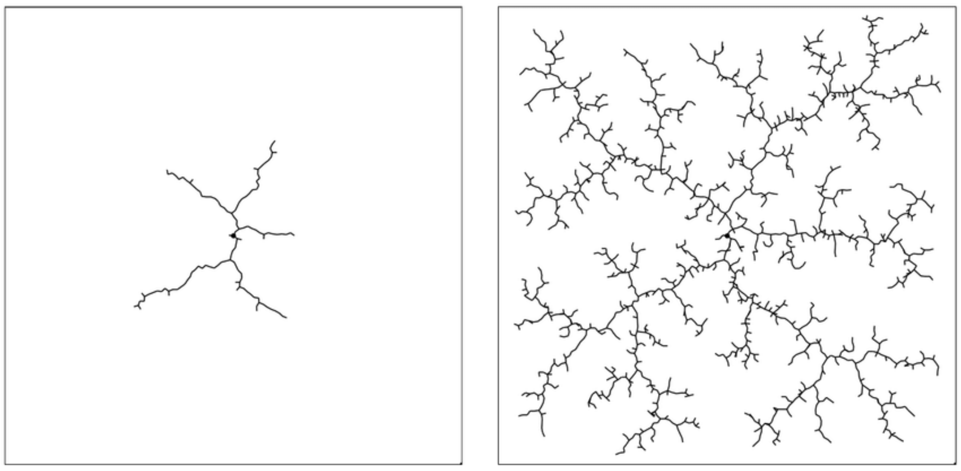


Figure 2.3.2: A visual representation of how RRT creates a tree that aims to decrease the distance between its created branches and an arbitrary point [34].

In addition to OMPL, Covariant Hamiltonian optimization for motion planning (CHOMP) and Stochastic Trajectory Optimization for Motion Planning (STOMP) are other MoveIt! motion planner options. They will be briefly explained, however, they are not used in this study. CHOMP Planner generates small trajectories that make the motion smoother and is used for trajectory optimization and collision avoidance. The planner comes with optimization parameters to adjust the planning time for setting the maximum time for finding the solution, as well as a parameter for collision clearance for setting the distance to be considered for collision free trajectories and more. In contrast to OMPL, CHOMP also checks for collisions instead of using external libraries to do so, however, OMPL is a faster planner for finding the trajectory. CHOMP can be used for post-processing the trajectory created by OMPL [11].

As for STOMP, it is a trajectory optimization planner that produces a noisy trajectory initially, then works on refining the solution found. An advantage for STOMP is that no postprocessing of the created trajectory is needed [72].

2.3.3. KINEMATICS HANDLING

Even though finding forward kinematics solutions is done in MoveIt! through the RobotState class without using plugins, inverse kinematics solutions need a solver. The RobotState class has information like position, velocity, and acceleration of the robot over time. The kinematic solver used in this study is *trac_ik_kinematics*. Where *trac_ik_kinematics* solver is a kinematic solver that uses threading to combine two inverse kinematics solving methods for more reliable results. This threading is done due to the limitations the inverse Jacobian method used by other solvers. Inverse Jacobian is a numerical method that is used to calculate inverse kinematics iteratively until the error decreases enough. This method, however, has no guarantee of convergence and does not work well in the presence of joint limits. *trac_ik_kinematics* expands on the inverse Jacobian method by using Sequential Quadratic Programming (SQP) optimization approach to work in the presence of joint limits. SQP is an iterative numerical method for non-linear constraints. Provided in Table 2.3.1 is data about the solving rate and time of different kinematics solvers, where *trac_ik* is seen to have the best performance [79]. Although the data is not provided for the e-series, it can be safely assumed that the same applies since the trend about the solver's performance is valid for all the robot models mentioned on the online GitHub repository for the solver [80].

Chain	DOFs	Orocos' *KDL* solve rate	Orocos' *KDL* Avg Time	*KDL-RR* solve rate	*KDL-RR* Avg Time	*TRAC-IK* solve rate	*TRAC-IK* Avg Time
Universal UR3	6	**17.11%**	4.18ms	**89.08%**	0.77ms	**98.60%**	0.45ms
UR5	6	**16.52%**	4.21ms	**88.58%**	0.74ms	**99.17%**	0.37ms
UR10	6	**14.90%**	4.29ms	**88.63%**	0.74ms	**99.33%**	0.36ms

Table 2.3.1: Kinematic solvers solve rate and time for different UR3, UR5, and UR10 [80].

The instructions for setting up the solver will be later mentioned in the Setup for Testing and Validation section in chapter 5.

2.3.4. TIME PARAMETERIZATION

MoveIt! is mainly a motion planning framework, it does not change the velocity and acceleration directly. Rather it uses time parameterization for velocity and acceleration control [78]. The velocity and acceleration of a joint are set to the default values in the robot descriptions in the joint_limits.yaml file that is read once upon launching ROS. These values can then be modified by setting them for each joint, or by changing the scaling factor for the velocity and acceleration using

ROS parameters. In the Fluxana Library, the primary way of changing the speed and acceleration is through setting the scale.

2.4. ROS VISUALIZATION (RVIZ)

RViz is used to visualize robot applications to show the robot's environment from the robot's perspective, therefore, it is a helpful tool in debugging. RViz uses transforms to calculate the relation between frames and displays them. RViz uses different plugins for different functionalities. Like the Robot Model plugin which displays the visual representation of the robot based on the URDF file, the TF plugin which displays the transformations of the robot frames, and the motion planning plugin which is used for trajectory planning and execution, etc. The robot's movement is restricted to, and cannot exceed its workspace, and the range of motions allowed is based on the planning group chosen. In RViz, there are four types of visualizations [49], which are:

- Robot's configuration in the planning scene
- Planned path
- Green starting state
- Orange goal state

Each axis in the coordinate system is indicated with a different color as follows:

- Red: x-axis
- Green: y-axis
- Blue: z-axis

The mentioned color codes can be seen in Figure 2.4.1:

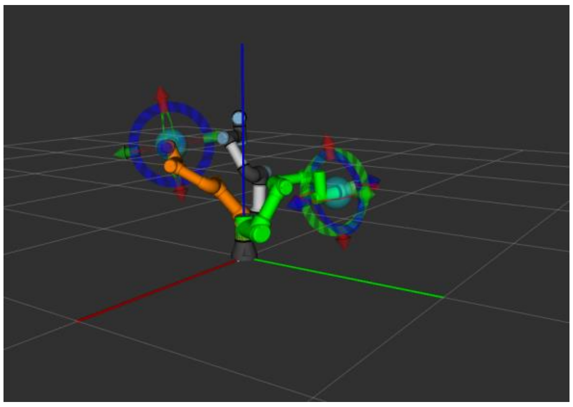
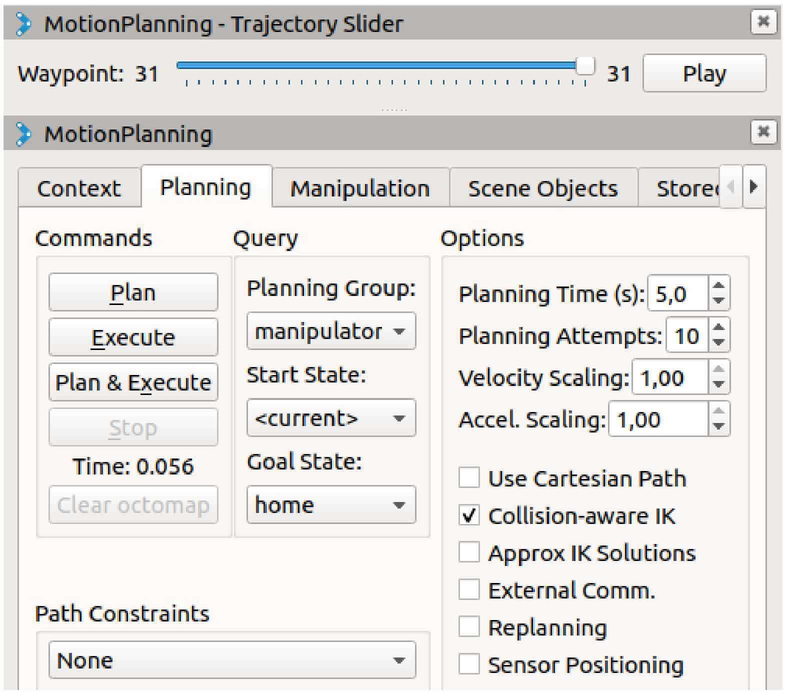


Figure 2.4.1: Illustration with the different color codes in RViz

To see the path the robot will take, the `Plan` command can be chosen. And to see intermediate steps as the robot moves from start to goal state, the option `Show Trail` can be chosen, where the trail size can be adjusted. This can be helpful in analyzing the path taken by the robot. The `Motion Planning Trajectory Slider` can then be used to rewind through the points taken during the motion planning. Different target can also be set from the Motion Planning tab along with parameter like velocity and speed scaling, where the plan can be shown in RViz before executing it on the real robot. The mentioned points can be seen in Figure 2.4.2:



Waypoint: 31

Time: 0.056

Planning Group: manipulator

Start State: <current>

Goal State: home

Planning Time (s): 5,0

Planning Attempts: 10

Velocity Scaling: 1,00

Accel. Scaling: 1,00

- [ ] Use Cartesian Path
- [✓] Collision-aware IK
- [ ] Approx IK Solutions
- [ ] External Comm.
- [ ] Replanning
- [ ] Sensor Positioning

Path Constraints: None

Figure 2.4.2: A snapshot from RViz with the Motion Planning plugin elements shown.

An essential concept for using MoveIt! and RViz is the URDF file, which provides a digital description for the robot and will be described in more detail in the next section.

2.5. UNIFIED ROBOT DESCRIPTION FORMAT (URDF)

URDF is an XML based language that is used to describe the robot using nested tags and contains information about the links, joints, and their limits, and more. This URDF file is essential to provide a digital description of the robot's kinematic chain for ROS applications such as MoveIt! and RVIZ. The units for the quantities used in the URDF files follow the International System of Units. URDF is used for basic robots, and does not allow including files like Xacro does, therefore, Xacro will be used to represent the UR3e in this study due to the robot's higher complexity. However, the format of a URDF file will be explained in this section. To describe joints and links the <joint> and <link> tags are used with the following sub-elements:

The joint description connects two links, where the child link is one whose pose depends on the parent link. The joint's origin is defined relative to the parent link reference frame. The joint definition contains the following elements:

- <origin>: transform from the parent link to the child link
- <parent>: defines the parent link
- <child>: defines the child link
- <axis>: the axis of rotation or translation based on the type of joint
- <limit>: defines the joint upper and lower limits

The joint type needs to be specified and can be as follows [53]:

- fixed: no degrees of freedom are offered
- continuous: can rotated without joint limits
- revolute: can rotate to angles from the lower bound to the upper. This joint type will be used to describe UR3e manipulator arm
- prismatic: moves translationally along an axis.

To describe the robot, information about the manipulator's visual appearance, collision, and inertia needs to be provided. The following sub-elements are used for providing this information [54]:

- <visual>: describes the robot's appearance
    - <origin>: visual element's reference frame
    - <geometry>: shape of the link
    - <material>: material of the link
- <collision>: used for collision detection. In some cases, it is the same as in the visual sub-element, but sometimes simpler geometries are used to save computational power.
    - <origin>: collision element reference frame
    - <geometry>: shape of the link for collision checking

  The following RViz snapshots show the visual description of the robot on the left and the collision description on the right. As can be seen, collision description uses more primitive shapes to save resources during collision checking. However, due to the complex nature of the UR3e, the following visualizations are created using Xacro files, which will be described in more details in the next section.

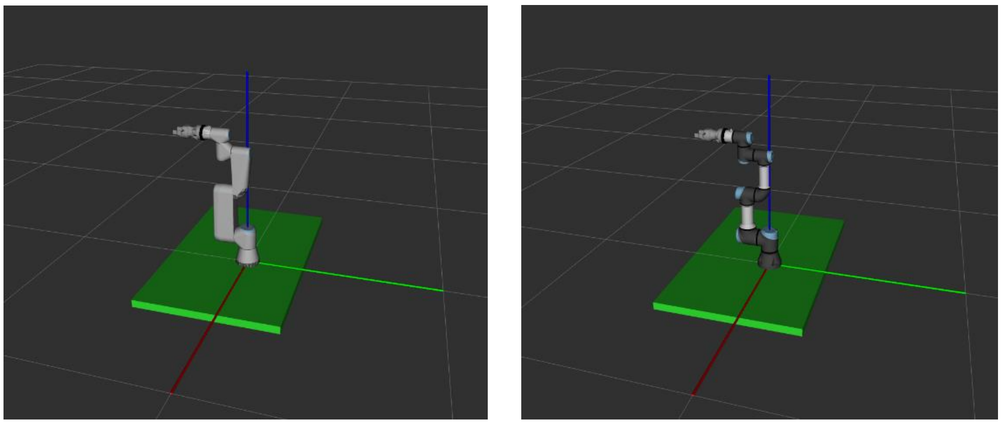


Figure 2.5.1: Collision link description (left) uses more primitive shapes than the visual link description (right).

- <inertial>: used to define quantities like the mass and its distribution
    - <origin>: center of mass frame
    - <mass>: mass of the link
    - <inertia>: link's moment of inertia

The table below lists some relevant UR3e parameters for defining the robot's digital description:

	Mass [kg]	Center of Mass
Link 1	1.98	[0, -0.02, 0]
Link 2	3.4445	[0.13, 0, 0.1157]
Link 3	1.437	[0.05, 0, 0.0238]
Link 4	0.871	[0, 0, 0.01]
Link 5	0.805	[0, 0, 0.01]
Link 6	0.261	[0, 0, -0.02]

Table 2.5.1: Mass and center of mass values for UR3e links [87].

2.6. XACRO

It is a macro language for XML that enhances the functionality of the URDF by allowing for code reuse and parameterization. Instead of adding the content of the file, the file itself can be included through its name, allowing for a more compact description that is easier to maintain as well as allows reusing files for different models without having to rewrite its contents. XACRO allows for the use of parameters and constants, which then gives the advantage of not having to repeat code [39]. For an excerpt of the used robot description in this study please refer to Appendix B: Xacro Code Sample for UR3e.

3. SYSTEM ARCHITECTURE

This chapter will break down how the system used in this study is structured to help gain understanding of the functionality of the components and concepts used as well as how they interact.

3.1. SYSTEM STRUCTURE

Since ROS provides full support only for Linux systems, a Linux machine is used for this study. On that machine there is a ROS system installed and set up using the “Ubuntu install of ROS Melodic” ROS documentation. MoveIt! Tutorials and ROS Wiki as well as the GitHub pages for Universal Robots ROS Driver, and those reference by ROS Wiki are important online sources for information and content.

After ROS is setup on the Linux machine, a workspace under the name catkin_ws is created. The catkin_ws contains different packages for the different applications, which will be mentioned in the file system section in this chapter. An example of these packages is the fx_library package itself, which contains functions that communicate with the robot through the Universal Robots ROS Driver over socket connections. The functions in the library use MoveIt!, and rospy functions. Some of these used functions also affect the environment in RViz either by changing the robot’s pose or by adding components to the planning scene for collision checking. To send and receive information from the robot or to control and actuate it, the external control URcaps program needs to be running to set the Linux machine up to allow it to send and receive information from the robot. For more information about how to do this set up, please refer to Appendix D: Universal Robots ROS Driver.

Another way communication with the robot was done in this study was through the polyscope. This was done for debugging purposes to check for discrepancies between the information received from the polyscope and that received from ROS. However, on the long term, the Fluxana Library aims to fully replace the polyscope and build Fluxana’s own API with functionalities needed for their applications. Figure 3.1.1 shows the system architecture used in this study:

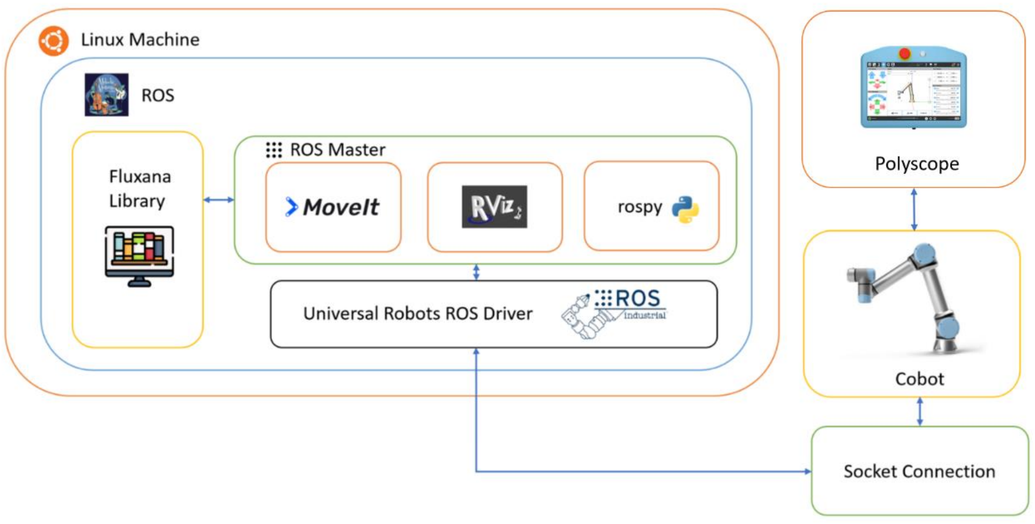


Figure 3.1.1: System architecture of the system used in this study [57][104][105][106][107][108][109][110].

3.2. FILE SYSTEM

As mentioned in the ROS chapter, to use ROS, there needs to be a workspace. catkin_ws is the one used for this study where it contains folders necessary for a workspace like the build, devel and src folders. In addition to that there’s a .git folder that links the local workspace to the online GitHub repository for backup.

Inside the src folder the packages used can be found. A main package is the fx_library package which has a structure aimed to make the library importable it should have the following directory tree shown in Figure 3.2.1.

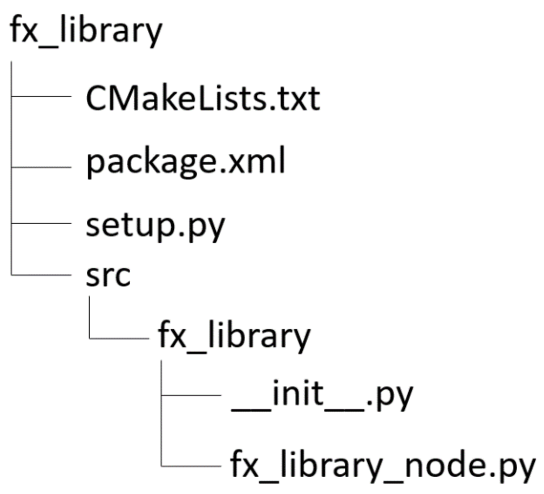


Figure 3.2.1: The file structure of the fx_library package

Where __init__.py is an empty python module used to mark the directory. This file structure will be discussed in more detail in the test setup section in chapter 4.

The testing is done through the test_fx_library package, where the test_fx_library.py file contains the unit tests for the Fluxana library functions. The Universal_Robots_ROS_Driver folder contains the launch file for the ROS Driver itself, and the ur3 package contains the launch files for MoveIt! and RViz. The commands to launch these launch files have been already documented in the launch file section in chapter 2.

Figure 3.2.2 gives an overview of the main files and folders used in this study:

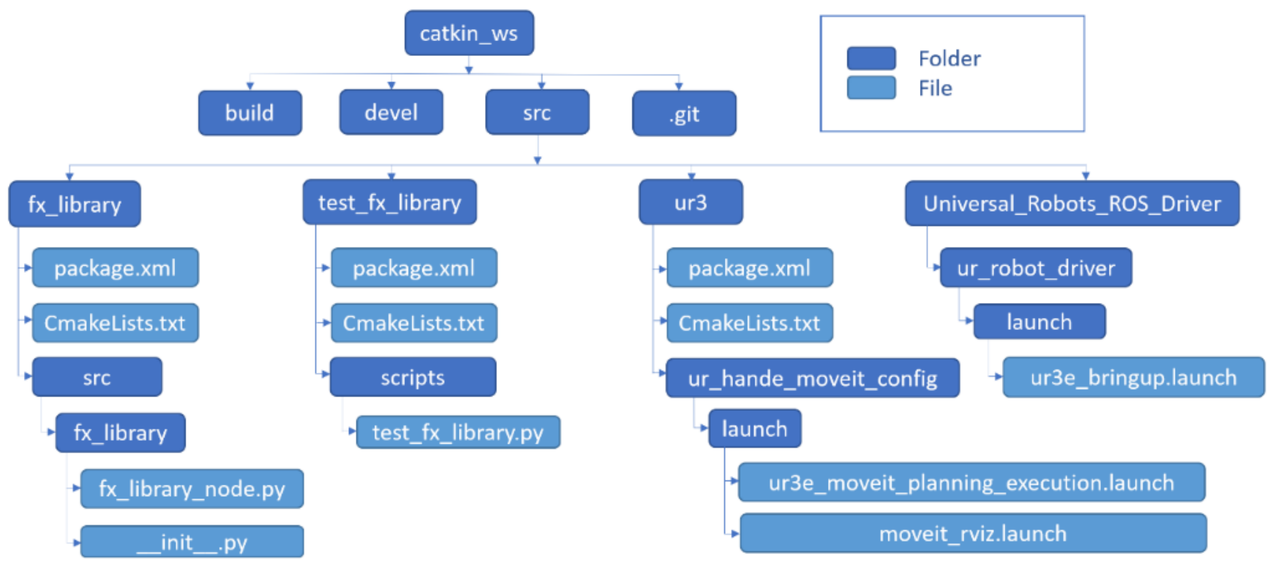


Figure 3.2.2: File structure of the catkin_ws used in the study.

4. FLUXANA LIBRARY

This chapter will discuss the Fluxana Library as described in the requirement specifications along with a guideline of how the functions were implemented. For better organization the functions will be divided into those that have to do with movement and functions that are needed to get the robot running, these will be called Kinematic Functions and General Functions respectively. The chapter will also explain helper functions that were added for convenience and cleaner code and avoiding code repetition. For more information about the ROS Services, Topics, and MoveIt! commands used please refer to Appendix L: Fluxana Library Explanation.

4.1. FUNCTIONS OF FLUXANA LIBRARY

The Fluxana Library functions can be divided into kinematic functions and general functions. Now each of the function groups will be documented in more detail.

4.1.1. KINEMATIC FUNCTIONS

An overview of the kinematic functions of the Fluxana Library can be found in the table below, where the description of each of the functions will follow.

Function	Page
Go to Home Position	36
Move Along an Axis	37
Move to Joint Values	37
Move to a Pose	37
Move from Start to End Pose	38
Stop Movement	38

Go to Home Position

This function is implemented in the Fluxana Library under the name *fx_go_home(velocity_factor = 1, acceleration_factor* = 1) and goes to the predefined home position with the joint values of $[0, \frac{-\pi}{2}, 0, \frac{-\pi}{2}, 0, 0]$ radians. The function has two arguments, one specifying the velocity scale and the other specifying the acceleration scale as a fraction in the range (0, 1]. The function implementation uses the Fluxana Library function *fx_move_joint(home_joint_position, velocity_factor = 1, acceleration_factor = 1)* to go to the target joint position and returns a boolean value specifying if the function execution failed or succeeded.

Move Along an Axis

This function is implemented in the Fluxana Library under the name $fx\_move\_distance(distance\_vector, velocity\_factor = 1, acceleration\_factor = 1)$ and is used to move a distance in meters along the x-, y-, and z- axes given by the 1x3 array argument specifying that distance. The other two parameter specify the velocity and acceleration scale as a fraction in the range (0, 1] as mentioned previously with the go home position function. The function returns a boolean indicating if the move command was executed successfully or not and uses the function $fx\_move\_pose(target, velocity\_factor = 1, acceleration\_factor = 1)$ in its implementation to move to the target pose.

Move to Joint Values

This function is implemented in the Fluxana Library under the name $fx\_move\_joint(target, velocity\_factor = 1, acceleration\_factor = 1)$ and is used to move the robot to the specified joint values argument in radians. The joint values are given in radians as a 1x6 array. The other two parameter specify the velocity and acceleration scale as a fraction in the range (0, 1] as with the other move commands discussed so far. The function returns a boolean value indicating if the move command was executed successfully or not. The implementation of the function uses two helper functions under the name $fx\_set\_velocity\_factor(velocity\_factor)$ and $fx\_set\_acceleration\_factor(acceleration\_factor)$ to set the velocity and acceleration scaling factors. It also uses the following functions from MoveIt! commander:

- $get\_current\_joint\_values()$: used to get the current joint values
- $set\_joint\_value\_target(target\_joint\_values)$: used to set the target joint values
- $go(wait = False)$: starts executing the move command asynchronously as specified by the wait = False parameter
- $stop()$: used to make sure there is no residual movement

Move to a Pose

This function is implemented in the Fluxana Library under the name $fx\_move\_pose(target, velocity\_factor = 1, acceleration\_factor = 1)$ and is used to move the robot to a specified pose, using either quaternion or Euler angle representation, with the position part of the pose is given in meters. The other two parameter specify the velocity and acceleration scale as a fraction in the range (0, 1] as with the other move commands discussed so far. The function returns a boolean value to indicate if the move command was executed successfully. The implementation of the function uses two helper methods under the names $fx\_set\_velocity\_factor(velocity\_factor)$ and $fx\_set\_acceleration\_factor(acceleration\_factor)$.

- $clear\_pose\_target(tcp)$: clears target poses before setting the new one
- $set\_pose\_target(target, tcp)$: sets the goal pose
- $go(wait = False)$: starts moving to target pose asynchronously

Move from Start to End Pose
This function is implemented in the Fluxana Library under the name $fx\_move\_relative(start\_pose, target\_pose, velocity\_factor = 1, acceleration\_factor = 1)$ and takes two pose arguments. One specifying the starting pose and the other specifying the target pose. Like the function to move to a pose, the angle representation can be either quaternion or Euler angles and the position part is defined in meters. The other two parameter specify the velocity and acceleration scale as a fraction in the range (0, 1] as with the other move commands. The function returns a boolean value specifying the move command was executed successfully and uses the Fluxana Library function $fx\_move\_pose(target, velocity\_factor, acceleration\_factor)$ in its implementation to move to the start and target poses.

Stop Movement
This function is implemented in the Fluxana Library under the name $fx\_stop()$ and is used to stop the robot from executing a move command if any is being executed. The function returns a boolean value indicating if the robot was successfully stopped and uses helper methods under the names: • $fx\_get\_max\_acceleration()$: gets the current maximum acceleration • $fx\_get\_acceleration\_scaling\_factor()$: gets the acceleraton scale • $fx\_get\_velocity()$: gets the current velocity These functions are used to calculate the appropriate time in which the robot will fully stop using the kinematic formula $vf = vi + at$ where the final velocity is zero and the initial velocity is the current velocity. The equation is then solved for the time variable, and a sleep command is issued to wait until the robot fully stops.

4.1.2. GENERAL FUNCTIONS

The following functions are necessary in getting the robot to start up or shut down as well as allow external control by the Linux machine, initialize the robot and more. The general functions are listed in the table below and will be described in this section. Some of the functions use the Dashboard Server response as a return value to the service, these responses can be found in Appendix C: Dashboard Server Commands.

Function	Page
Power On	39
Release Brakes	39
Power Off	40
Get Robot Mode	40
Start External Control	40
Stop External Control	41
Initiatialize the Robot	41

Power On

This function is implemented in the Fluxana Library under the name $fx\_power\_on()$ and is used to power on the robot. Upon starting up the robot it is initially in “POWER_OFF” mode. This mode can also be achieved with the power off function. To start using the robot, it needs to be in “RUNNING” mode, which can be achieved by the brake release function. Powering on the robot is an intermediate step before releasing the robot’s brakes and changes its mode to “IDLE”. The function to power on the robot uses the ROS Service under the name $/ur\_hardware\_interface/dashboard/power\_on$ which has the type $stdsrvs/Trigger$. The $message$ field of this service message is used to get the dashboard server response and add it to logs and the $success$ field will be used to indicate if the function executed successfully. The function checks for successful execution by using the $fx\_get\_robot\_mode()$ to see when the robot mode changes to “IDLE”, upon which the function returns a boolean value with the execution status.

Release Brakes

This function is implemented in the Fluxana Library under the name $fx\_brake\_release()$ and is used to release the robot’s brakes upon powering it on. The implementation of the function uses the ROS Service for releasing the brakes under the name $/ur_hardware_interface/dashboard/brake$_release. This service has a type of $std\_srvs/Trigger$ and contains a String field called $message$ containing the dashboard server response which is added to logs in the function implementation. The other used field is a boolean field named $success$ that indicates if the command was sent successfully, similarly with the powering on function. The execution status is returned upon changing of the robot’s mode to “RUNNING”. To continuously check when the robot mode changes, the function $fx\_get\_robot\_mode()$ is used. For more information about how this function is related to the power on and power off functions, please refer to the power on function documentation.

Power Off

This function is implemented in the Fluxana Library under the name $fx\_power\_off()$ and is used to power off the robot. This means that to use the robot again, one needs to power it on and release the brakes. For the corresponding modes of each of those actions, please refer to the powering on function documentation. It is important to differentiate this function from the shutting down function which shuts down the robot and control box. The function uses the ROS Service to power off the robot under the name $/ur\_hardware\_interface/dashboard/power\_off$ which has the type $stdsrvs/Trigger$ like with the release brakes function the $message$ field is used to get the dashboard server response and add it to logs and the $success$ field will be used to indicate if the function executed successfully. The difference from the power on and brake release functions would be that the fx_get_robot_mode() will check that the robot mode changes to “POWER_OFF”.

Get Robot Mode

This function is implemented in the Fluxana Library under the name $fx\_get\_robot\_mode()$ and is used to get the robot’s mode with the most relevant modes being “IDLE”, “POWER_OFF”, and “RUNNING”. For more information about other modes, please refer to Appendix C: Dashboard Server Command. These modes are used to determine if the robot has been successfully powered on, powered off, or has its brakes released. The function implementation uses the ROS Service $/ur\_hardware\_interface/dashboard/get\_robot\_mode$ of the type $ur\_dashboard\_msgs/GetRobotMode$ to get the robot mode. The service message contains a field named $answer$ which contains a string with the robot’s mode in the form of "Robotmode: " plus a string containing the mode. The value returned is only that of the mode without the added "Robotmode: ".

Start External Control

This function is implemented in the Fluxana Library under the name $fx\_start\_external\_control()$ and is used to start the $external\_control.urp$ program which specifies that the Linux machine used for this study with the IP address of 192.168.11.1 can be used to control the robot remotely. The function is divided into two parts one that loads the $external\_control.urp$ program using the ROS service $/ur\_hardware\_interface/dashboard/load\_program$ of type $ur\_dashboard\_msgs/Load$ to load it. The service message has the field $success$, which is checked to see if the program was loaded and displays the dashboard server message contained in the field $answer$ upon failure. The second part of the function runs the loaded program using the ROS service $/ur\_hardware\_interface/dashboard/play$ of type $std\_srvs/Trigger$, which has a field named $message$ containing the dashboard server response. The content of this field is then added to logs and the boolean field $success$ will be returned to indicate if the program has been started. The function implementation uses a helper function under the name $fx\_get\_program\_state()$ to get whether the robot has a running program.

Stop External Control

This function is implemented in the Fluxana Library under the name $fx\_stop\_external\_control()$ and is used to stop running programs on the robot. It is used for when it is required to stop external control mode. The function implementation used the ROS Service $/ur\_hardware\_interface/dashboard/stop$ of type $std\_srvs/Trigger$ to stop the program and logs the $message$ field containing the dashboard server response. The function then checks if the program was stopped using the helper function $fx\_get\_program\_state()$ and returns if the program was stopped successfully.

Initiatialize the Robot

This function is implemented in the Fluxana Library under the name $fx\_init()$ and is meant to contain the necessary functions needed to prepare the robot for initial operation. This includes powering on the robot using the $fx\_power\_on()$ function, and releasing its brakes using the function $fx\_brake\_release()$ if not already done. The state of the robot is checked to be in "RUNNING" mode using the function $fx\_get\_robot\_mode()$. In order to control the robot, the $external\_control.urp$ program needs to be running, so this function starts the external control using the function $fx\_start\_external\_control()$. This function also adds a box into the planning scene that represents the table on which the robot is mounted on for the purpose of collision checking.

Get Remote Control Status

This function is implemented in the Fluxana Library under the name $fx\_is\_in\_remote\_control()$ and is used to get whether the robot is in remote control. The function implementation uses the ROS Service $/ur\_hardware\_interface/dashboard/is\_in\_remote\_control$ of type $ur\_dashboard\_msgs/IsInRemoteControl$ to get the remote control status. The service message contains a field named $in\_remote\_control$ which contains a boolean with the answer and is used as the return value of the function.

Shut Down

This function is implemented in the Fluxana Library under the name $fx\_shutdown()$ and is used to shut down the robot and the controller. The function uses the ROS service $/ur\_hardware\_interface/dashboard/shutdown$ of type $std\_srvs/Trigger$ that contains the $message$ field containing the response from the dashboard server. This field is then added to the logs. The message also contains a boolean field $success$ indicating if the command was sent successfully. This boolean field is returned by the function.

Unlock Protective Stop
This function is implemented in the Fluxana Library under the name $fx\_unlock\_protective\_stop()$ and is used to unlock the robot's protective stop in case the robot accelerates too quickly from a low velocity to a higher one, if the robot's end effector has a collision, or if the controller detects a problem [84]. The function implementation uses the ROS Service $/ur\_hardware\_interface/dashboard/unlock\_protective\_stop$ of type $std\_srvs/Trigger$ to unlock the protective stop. The service type contains a string field called $message$ that returns the dashboard server response and is added to logs. The boolean field $success$ returns whether the command was sent, and is retuned by the function.

4.2. ADDITIONAL FUNCTIONS USED IN FLUXANA LIBRARY

The following are helper methods which are not required for the Fluxana Library but are added for convenience and for avoiding repeating code.

Function	Page
Call a Service	42
Subscribe to Topic One Time	43
Get the Acceleration Scaling Factor	43
Set Acceleration Scaling Factor	43
Get Velocity Scaling Factor	43
Set Velocity Scaling Factor	43
Get the Maximum Acceleration	44
Get the Program Status	44
Get Current Velocity	44

Call a Service
This function is implemented under the name $fx\_call\_service(service\_name, srv, arg1 = None, arg2 = None, arg3 = None)$ and is used to call a ROS service with up to three arguments. The function uses the Fluxana Library function $fx\_is\_in\_remote\_control()$ to check if the robot is in remote control before calling the ROS service. In the case that remote control mode is on, the service is called, and the response of the service is returned. There is no specific type that the function returns as the response of different services has different types. It is therefore important to see the documentation for the service type to know what the returned quantity is. For this study, the list of used services and their types is in the Appendix L: Fluxana Library Explanation, for more information please refer to that section.

Subscribe to Topic One Time

This function is implemented under the name $fx\_subscribe\_topic\_once(topic\_name, msg\_type)$ and is used to subscribe to a topic and retrieve the published information once before returning to normal program excution. Before subscribing to the topic, the function checks if the robot is in remote control mode using the Fluxana Library function $fx\_is\_in\_remote\_control()$ and if remote control mode is on, it returns the information returned from calling the $rospy$ function $rospy.wait\_for\_message(topic\_name, msg\_type, timeout = None)$ which is equivalent to subscribing to a topic and receiving the published information one time. Like with the calling the service function, there is no specific datatype that the function returns as different topics have different types.

Get the Acceleration Scaling Factor

This function is implemented under the name $fx\_get\_acceleration\_scaling\_factor$() and is used to get the value of the ROS parameter $default\_acceleration\_scaling$. The returned value is a fraction in the range (0, 1].

Set Acceleration Scaling Factor

This function is implemented under the name $fx\_set\_acceleration\_factor(acceleration\_factor)$ and is used to set the value for the ROS parameter $default\_acceleration\_scaling\_factor$. The allowed values are on the range of (0, 1]. The function returns a boolean indicating if the parameter was changed to the value in the argument.

Get Velocity Scaling Factor

This function is implemented under the name $fx\_get\_velocity\_scaling\_factor()$ and is used to get the ROS parameter $default\_velocity\_scaling\_factor$ located in the $joint\_limits.yaml$. The returned value is a fraction in the range (0, 1].

Set Velocity Scaling Factor

This function is implemented under the name $fx\_set\_velocity\_factor(velocity\_factor)$ and is used to set the value for the ROS parameter $default\_velocity\_scaling\_factor$. The allowed values are on the range of (0, 1]. The function returns a boolean indicating of the value in the argument was set for the parameter.

Get the Maximum Acceleration
This function is implemented under the name $fx\_get\_max\_acceleration()$ and is used to get the ROS parameters for the joints' maximum accelerations. The returned value is a 1x6 array containing the defined maximum acceleration from the corresponding ROS parameters, for more information about the names of the parameters, please refer to the function implementation Appendix L: Fluxana Library Explanation.

Get the Program Status
This function is implemented under the name $fx\_get\_program\_state()$. The function implementation uses the ROS service $/ur\_hardware\_interface/dashboard/program\_state$ of the type $ur\_dashboard\_msgs/GetProgramState$ which contains a string field $state$ with either "PLAYING" or "STOPPED" based on whether a program is running on the robot as well as a $program\_name$ field containing what program is being run. The function returns the value for the service message $ur\_dashboard\_msgs/GetProgramState$.

Get Current Velocity
This function is implemented under the name $fx\_get\_velocity()$ and it uses the topic $/joint\_states$ which has information about the state of the joints constantly being published to it including information about the joint's velocity and position. The topic has the type $sensor\_msgs/JointState$, and the returned value from that function is the $velocity$ field of the message, which is a 1x6 array of double containing the velocity of the robot at the moment the topic is subscribed to. The function uses another helper function under the name $fx\_subscribe\_topic\_once(topic\_name, msg\_type)$ to get published information from the topic only once.

4.3. ADDITIONAL FUNCTIONS USED IN TESTING FLUXANA LIBRARY

Function	Page
Get Current Joint Values	46
Get Current Pose with Quaternion Angles	46
Get Current Euler Angles	46
Get Current Pose with Euler Angles	46

Get Current Joint Values

This function is implemented under the name $fx\_get\_current\_joint\_values()$ and is used to return the joint values of the robot at the time the function is being called. This data is returned in the form of a 1x6 array of double containing the values for each of the joint in radians using the function get_current_joint_values() from the MoveIt! framework.

Get Current Pose with Quaternion Angles

This function is implemented under the name $fx\_get\_current\_pose(end\_effector\_link = tcp)$ and is used to get the current pose of the specified frame, where by default it is the TCP of the robot. The units for the position are given in meters and the orientation representation is quaternion. The $get\_current\_pose(end\_effector\_link)$ function from the Moveit! framework is used to get the value that the function returns, which is an instance of the $geometry\_msgs.msg$.

Get Current Euler Angles

This function is implemented under the name $fx\_get\_current\_rpy(end\_effector\_link = tcp)$ is used to get the Euler angles for the specified frame, by default, given as the robot's TCP. The function $get\_current\_rpy(end\_effector\_link)$ from the Moveit! Framework is used to get the 1x3 array of double containing the Euler angles.

Get Current Pose with Euler Angles

This function is implemented under the name $fx\_get\_current\_pose\_rpy()$ and is used to get the current pose of the TCP with the position given in meters and the Euler angle representation. The function uses the helper functions $fx\_get\_current\_pose()$ to get the current pose in quaternion angle representation and $fx\_get\_current\_rpy()$ to get the current Euler angles.

For more information about the Fluxana Library, please refer to Appendix L: Fluxana Library Explanation.

5. TESTING AND VALIDATION

In this chapter, functions will be tested to see if they behave as expected. First the software and hardware setup used for the experiment will be described, then the testing will take place. The testing is done using two methods. One is by logging relevant data for analysis of function performance and the second is by using Unit Testing the functions to make sure the function runs correctly. Firstly, a series of tests will be done initially to check if the URDF representation of the robot corresponds to the actual robot before proceeding. The kinematic functions will then be tested to check if the robot goes to a specified pose with a high precision and accuracy, and that the stop function stops ongoing move commands, as well as test if the functions to set the velocity and acceleration scales work as expected. The general functions will then be tested by making sure that the functions to release brakes, power on, and power off the robot achieve the expected outcome and that external control program is run and stopped correctly. Now, the test setup will be described, then a more in detail description of the tests will follow.

5.1. SETUP FOR TESTING AND VALIDATION

The hardware and software listed below is used in testing the Fluxana Library. Some relevant information about the installation and setup is included in this section or in the corresponding Appendix.

5.1.1. ROS MELODIC

The ROS Melodic distribution is chosen for this study based on what version is compatible with the Niryo Ned 2 which is used for another study in parallel. ROS Melodic is a ROS 1 version that has Long Term Support (LTS). For instructions on how to install ROS Melodic please visit the ROS WIKI page on “Ubuntu install of ROS Melodic”. Since ROS Melodic supports python2 and not python3, python 2.7.17 is used for coding in this study [111].

5.1.2. UBUNTU 18.04.6

Ubuntu is the main operating system used with ROS and for the same reason of compatibility with Niryo Ned 2, which uses Ubuntu 18.04 LTS release. The updated version Ubuntu 18.04.6 is used for this study.

5.1.3. LINUX MACHINE SPECIFICATIONS

At the beginning of the study the Linux Machine used would crash often whenever RViz was run as well as had a very slow speed in general. Therefore, the machine was changed to another one with a more powerful processor which allowed for more comfortable multitasking while using it. The graphics card in the new Linux machine also solved the problem of crashing whenever RViz was used. Another reason the new computer had improved performance when working with RViz is that it has a dedicated Graphics Processing Unit (GPU) NVIDIA GEFORCE, which is a high-performance GPU.

The specifications of the old and new Linux machines are shown below:

Old	Processor: intel Celeron(R) CPU P4600 @2.00GHz x2 Graphics: intel HD Graphics (ILK) GPU: None OS Type: 64-bit
New	Processor: intel core i7-4500U CPU @ 1.80GHz x4 Graphics: Intel HD Graphics 4400 (HSW GT2) GPU: NVIDIA GEFORCE OS type: 64-bit

5.1.4. UNIVERSAL ROBOTS UR3E

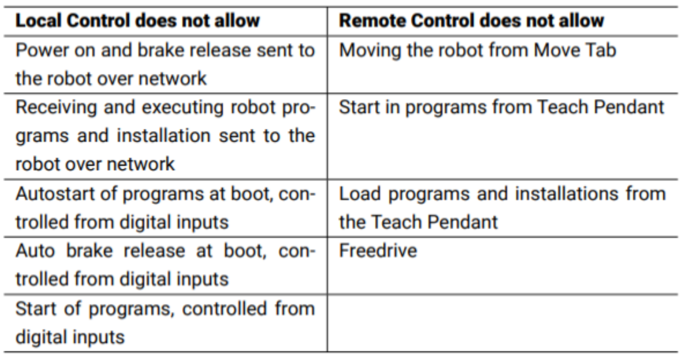
The Universal Robot used in this study is the UR3e model and has the IP address 192.168.11.100. The robot comes with a teach pendant that shows the robot's information and takes commands from the used, which was used for double checking the performance of the library functions during programming. The software version used on that teach pendant is 5.11.5.1010327 (Oct 15 2021).

The UR3e can carry a workload of up to 3kg. It has reach of 500mm and repeatability of $\pm 0.03mm$. The robot has six revolute joints, meaning that it had six degrees of freedom. Each joint has a defined working range and angular speed as follows, for more information about the robot's specifications please refer to Appendix A: UR3e Technical Details.

Joint	Working Range	Maximum Speed [°/s]	Maximum Torque [Nm]
Base	$\pm 360°$	± 180	56
Shoulder	$\pm 360°$	± 180	56
Elbow	$\pm 360°$	± 180	28
Wrist 1	$\pm 360°$	± 360	12
Wrist 2	$\pm 360°$	± 360	12
Wrist 3	Infinite	± 360	12

Table 5.1.1: Joint limits for UR3e [83][90].

The cobot comes with two modes of control, local and remote. Remote control is used for the application since not all functions in the Fluxana Library can be executed in local control mode. These functions include but are not limited to, releasing the brakes, powering the robot on, running, and stopping programs. As a safety feature, Universal Robots only allow switching to remote control through the teach pendant, therefore, the robot will be initially put on remote control before using the library functions. The following image from the robot specifications shows the limitations of each mode of control.



Local Control does not allow	Remote Control does not allow
Power on and brake release sent to the robot over network	Moving the robot from Move Tab
Receiving and executing robot programs and installation sent to the robot over network	Start in programs from Teach Pendant
Autostart of programs at boot, controlled from digital inputs	Load programs and installations from the Teach Pendant
Auto brake release at boot, controlled from digital inputs	Freedrive
Start of programs, controlled from digital inputs	

Figure 5.1.2: Local and remote control mode limitations [85].

To use ROS with the robot, Universal Robots ROS Driver needs to be installed. For instructions on how to set it up, please refer to Appendix D Universal Robots ROS Driver. Installing the driver automatically sets up a MoveIt! package and RViz. This automatic setup, however, does not include the gripper. Therefore, a manual setup is done for this study to consider the gripper in the collision checking. For more information please refer to Appendix E: MoveIt! Setup Assistant.

5.1.5. MOVEIT! FRAMEWORK

Setting up MoveIt! is essential for the library as it serves as a basis for the kinematic functions in the library. To set up the package for the UR3e, please refer to Appendix E: MoveIt! Setup Assistant. Some of the configuration files like joint_limit.yaml and parameters like tolerance and kinematic solver specifications used in MoveIt! will be explored in some of the tests to see if the performance can be enhanced.

5.1.6. SETTING UP THE INVERSE KINEMATICS SOLVER

To install trac_ik inverse kinematic solver use the following command in the Linux terminal [79]:

```
sudo apt-get install ros-melodic-trac-ik-kinematics-plugin
```

- In the kinematics.yaml file update the kinematic solver parameter

```
kinematics_solver: trac_ik_kinematics_plugin/TRAC_IKKinematicsPlugin
```

The Kinematic solver parameters for the resolution and timeout are set by default to 0.005, however, testing will be done to see if more optimal values can be used to better suit the application.

5.1.7. SETTING UP FLUXANA LIBRARY AS A PYTHON MODULE

To test the code, a test package under the name test_fx_library will import the module fx_library_node from the fx_library package and run the tests on it. To achieve this the fx_library package should have the structure discussed in the File System Section in chapter 3. For the contents of the setup.py file and required changes for the package.xml and CMakeLists.txt in both the library and test packages refer to ros.org documentation for “Configuring rostest”.

Now that the test set up has been explained, next will be explained how in this study the URDF was matched with the actual robot to get correct results for move commands in ROS, as well as how adding a virtual table to the planning scene for collision avoidance is done.

5.2. PREPARATION FOR TESTING AND VALIDATION

An issue that was faced during executing trajectories was that the robot would have a time out during execution of given poses and joint values and an error would be issued mid execution. This was due to a timeout set for move command execution. This allowed goal duration margin was set to 0.5s, so it was changed to 60s to allow for more time to execute the command. The parameter mentioned can be seen below:

trajectory_execution.launch.xml

```
<launch>
…
     <param name="trajectory_execution/allowed_goal_duration_margin" value="60"/>
...
</launch>
```

Now that move commands can be executed, it is tested if they are executed correctly in the next section.

5.2.1. TESTING GOING TO A POSE

Another error occurred whenever a pose was given to MoveIt! to execute, where the robot would go to a different pose rather than the one corresponding to the coordinate system used in this study as described in the Kinematic Chain section in chapter 2. It was made clear that there is a discrepancy between the actual robot and the URDF or the representation of the orientation. To confirm that both models are not equivalent, the frame for the links was retrieved using the fx_get_current_pose(end_effector_link), where the end-effector link is chosen arbitrarily as “wrist_2_link” for validation purposes. The following approximated values were received:

Link	Position	Orientation in Quaternion	Orientation in Euler
"wrist_2_link"	x: 0.005992 y: 0.131062 z: 0.693904	x: -0.006164 y: 0.0061952 z: -0.703950 w: 0.710195	x: -0.01758393 y: 0.0001210 z: -1.561930

Using Rviz to visualize the "wrist_2_link" frames from the TF tab, the following figure is shown:

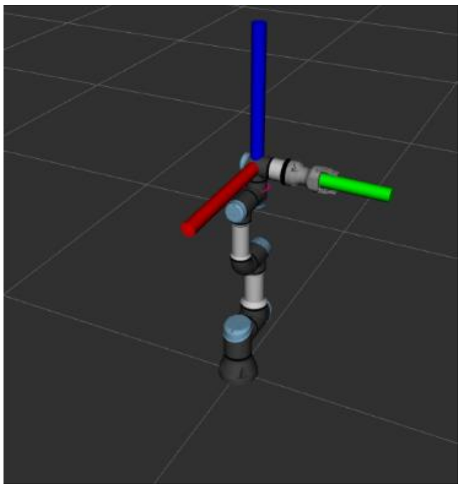


Figure 5.2.1: Wrist 2 frame in RViz

The wrist_2_link frame is created in the polyscope, and the point created has x- and y- axes that are in the opposite direction to the figure in RViz and does not overlap with the robot. It is important to note that there is a second point displayed by default for the TCP in the figure below, however, it is not needed in this analysis:

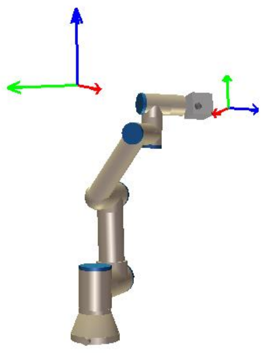


Figure 5.2.2: Shows the wrist_2_link pose value from ROS on the polyscope for debugging purposes.

To test how the coordinates do not correspond, the following test pose is taken at random:

Test Pose using Rotation Vector Angles
[-0.32612, -0.25633, 0.64783, 1.545, -0.647 -0.846]

Since the polyscope uses rotation vector orientation representation, the orientation in this pose will be converted to Euler representation:

Test Pose using Euler Angles
[-0.32612, -0.25633, 0.64783, 1.611, 0.151, -0.945]

This pose corresponds to the following robot configuration, where the negative y-axis can be seen to come out of the page and the negative x-axis is towards the left, and positive z-axis pointing upwards.

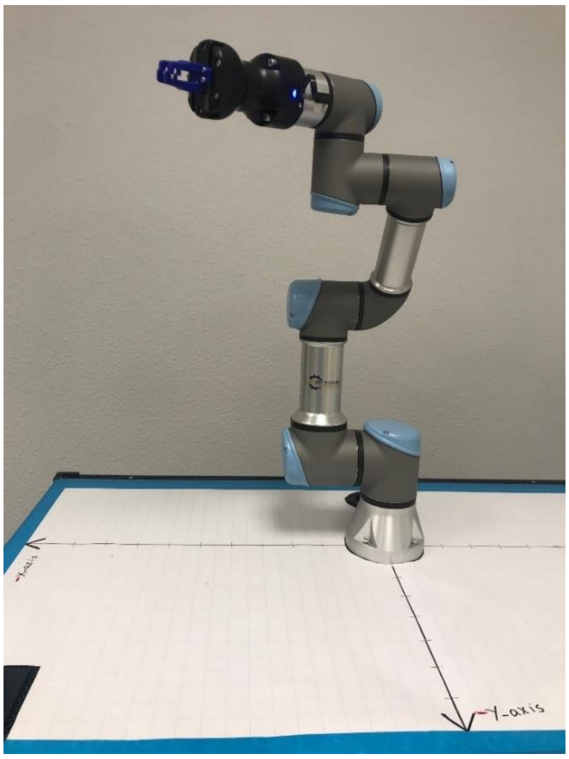


Figure 5.2.3: The robot at pose [-0.32612, -0.25633, 0.64783, 1.611, 0.151, -0.945]

Using MoveIt! to get the current pose yields:

position	Quaternion orientation	Orientation in Euler
x: 0.325637556736	x: 0.260516127664	X: 1.5974348857895226
y: 0.258305620703	y: 0.667241106324	Y: 0.1894805581810313
z: 0.648814494734	z: 0.590550546134	Z: 2.21259294909246
	w: 0.371713203586	

Which shows already that the x- and the y- axes have opposite signs than expected. There is also a discrepancy between the angles displayed on the robot and the Euler equivalent of the angles from movement. Reasons for the problem can be either a different orientation of the frames in the URDF, but another possible reason could be that the polyscope uses Euler angle in a different combination than ROS. And since rotations are not commutative this leads to results that do not match. It is therefore important to locate the source of this error.

Universal Robots uses Euler angles in configuration RzRyRx, however, to check that maybe another combination yields the quaternion angle of [0.260516127664, 0.667241106324 , 0.590550546134, 0.371713203586], the Euler angles [1.611, 0.151, -0.945] will be converted to quaternion using the transformations.quaternion_from_euler function that takes an argument specifying the Euler angle and another with the rotation combination. The following table shows the resulting quaternion angle for the different combinations used:

Euler Combination	Quaternion
R_{zyx}	[-0.36281929, -0.28075481, 0.6641118 , 0.59033416]
R_{yxz}	[-0.28075481, 0.6641118 , -0.36281929, 0.59033416]
R_{xzy}	[0.6641118 , -0.36281929, -0.28075481, 0.59033416]
R_{xyz}	[0.61654967, 0.37381044, -0.26594521, 0.63984795]
R_{yzx}	[-0.26594521, 0.61654967, 0.37381044, 0.63984795]
R_{zxy}	[0.37381044, -0.26594521, 0.61654967, 0.63984795]

Table 5.2.1: Quaternion results for different Euler combinations of angles [1.611, 0.151, -0.945].

Although all conversions lead to values that are reasonably close to the given quaternion representation, the order is not correct. Therefore, the Euler angle configuration is not the problem. Another possible problem is incorrect in the URDF representation of the real robot.

Looking at the robot in RViz with all joint values of [0, 0, 0, 0, 0, 0] and corresponding pose of [-0.45658, -0.36934, 0.06209, 1.547, 1.514, -0.033], to use the DH parameters table from the DH parameters section in chapter 2. From the following figure, it can be seen that some frames have different orientations than the one specified by the kinematic chain description for DH parameters:

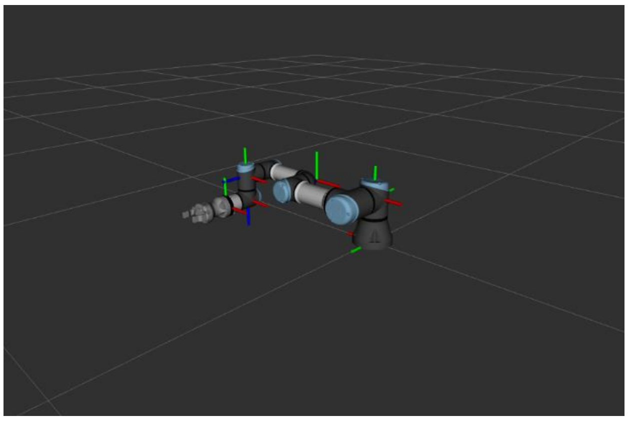


Figure 5.2.4: The transforms of the robot frames at joint angles [0, 0, 0, 0, 0, 0] in RViz

The following changes were made to the ur_macro.xacro file used for the MoveIt! package to get the digital representation of the robot to match the real robot. These changes were done by analyzing the kinematic chain from the base to the tool tip and making changes in the parameters by debugging with RViz, the change made were as follows:

Tag	`<link name="${prefix}upper_arm_link">`
Old	`<origin xyz="0 0 ${shoulder_offset}" rpy="${pi/2} 0 ${-pi/2}"/>`
New	`<origin xyz="0 0 ${shoulder_offset}" rpy="${-pi / 2} ${pi} ${pi / 2}"/>`

Tag	`<link name="${prefix}wrist_1_link">`
Old	`<origin xyz="0 0 ${wrist_1_visual_offset}" rpy="${pi/2} 0 0"/>`
New	`<origin xyz="0 ${wrist_1_visual_offset} 0" rpy="0 0 0"/>`

Tag	`<link name="${prefix}wrist_2_link">`
Old	`<origin xyz="0 0 ${wrist_2_visual_offset}" rpy="0 0 0"/>`
New	`<origin xyz="0 ${-wrist_2_visual_offset} 0" rpy="${pi / 2} 0 0"/>`

Tag	`<joint name="${prefix}base_link-base_link_inertia" type="fixed">`
Old	`<origin xyz="0 0 0" rpy="0 0 ${pi}" />`
New	`<origin xyz="0 0 0" rpy="0 0 0" />`

Tag	`<joint name="${prefix}shoulder_pan_joint" type="revolute">`
Old	`<origin xyz="${shoulder_x} ${shoulder_y} ${shoulder_z}" rpy="${shoulder_roll} ${shoulder_pitch} ${shoulder_yaw}" />` `<axis xyz="0 0 1" />`
New	`<origin xyz="${shoulder_x} ${shoulder_y} ${shoulder_z}" rpy="${shoulder_roll + pi / 2} ${shoulder_pitch} ${shoulder_yaw}" />` `<axis xyz="0 1 0" />`

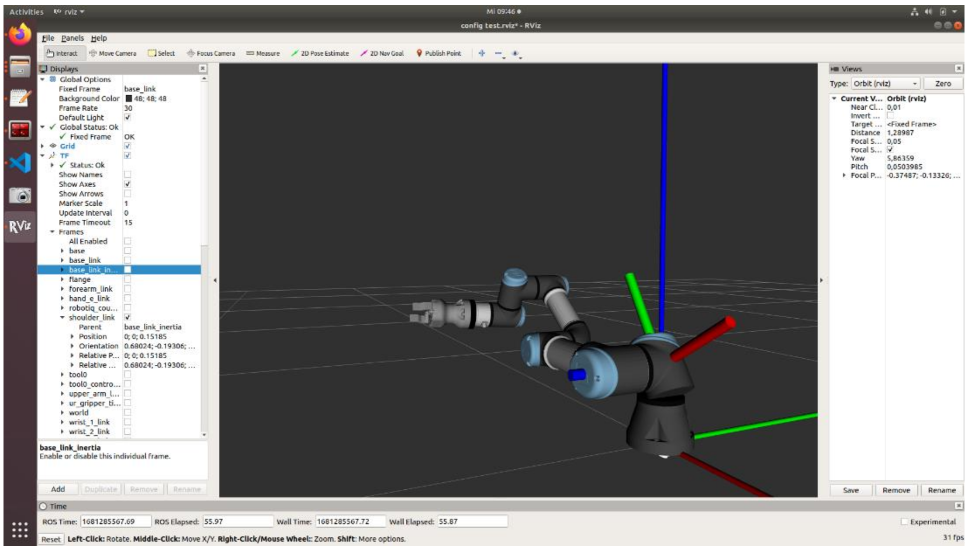
Tag	`<joint name="${prefix}shoulder_lift_joint" type="revolute">`
Old	`<origin xyz="${upper_arm_x} ${upper_arm_y} ${upper_arm_z}" rpy="${upper_arm_roll} ${upper_arm_pitch} ${upper_arm_yaw}" />`
New	`<origin xyz="${upper_arm_x} ${upper_arm_y} ${upper_arm_z}" rpy="${upper_arm_roll + 3 * pi / 2} ${upper_arm_pitch} ${upper_arm_yaw}" />`

Tag	`<joint name="${prefix}wrist_1_joint" type="revolute">`
Old	`<origin xyz="${wrist_1_x} ${wrist_1_y} ${wrist_1_z}" rpy="${wrist_1_roll} ${wrist_1_pitch} ${wrist_1_yaw}" />` `<axis xyz="0 0 1" />`
New	`<origin xyz="${wrist_1_x} ${wrist_1_y} ${wrist_1_z}" rpy="${wrist_1_roll + pi / 2} ${wrist_1_pitch} ${wrist_1_yaw}" />` `<axis xyz="0 1 0" />`

Tag	`<joint name="${prefix}wrist_2_joint" type="revolute">`
Old	`<origin xyz= "${wrist_2_x} ${wrist_2_y} ${wrist_2_z}" rpy="${wrist_2_roll } ${wrist_2_pitch} ${wrist_2_yaw}" />` `<axis xyz="0 0 1" />`
New	`<origin xyz="0 0 0.08535" rpy="${wrist_2_roll + pi} ${wrist_2_pitch} ${wrist_2_yaw}" />` `<axis xyz="0 -1 0" />`

Tag	`<joint name="${prefix}wrist_3_joint" type="revolute">`
Old	`<origin xyz= "${wrist_3_x} ${wrist_3_y} ${wrist_3_z}" rpy="${wrist_3_roll} ${wrist_3_pitch} ${wrist_3_yaw}" />`
New	`<origin xyz="${wrist_3_x} 0 0.0921" rpy="${wrist_3_roll + pi / 2} ${wrist_3_pitch} ${wrist_3_yaw}" />`

The changes in the axis of rotations were made since changing the orientation of the frame meant that the axis around which the joint revolved needs to be redefined. For example, trying to rotate the base link in the positive direction should cause circular motion around the z-axis of the world or the blue extended axis in the following figure. Doing the following rotation without making the changes, however, leads to the incorrect results as shown below.



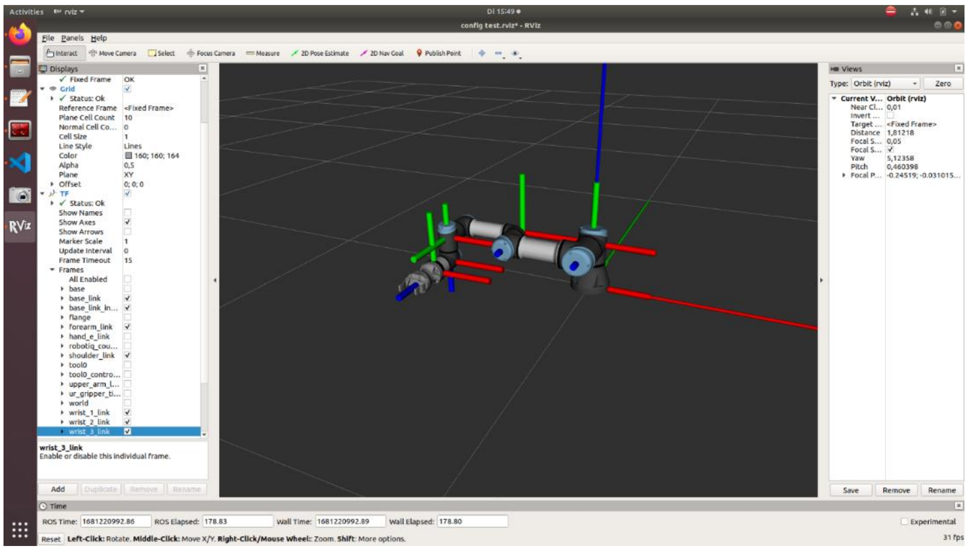
```
Global Options
  Fixed Frame          base_link
  Background Color     48; 48; 48
  Frame Rate           30
  Default Light        ✓
✓ Global Status: Ok
  ✓ Fixed Frame        OK
Grid                   ✓
TF                     ✓
  ✓ Status: Ok
  Show Names
  Show Axes            ✓
  Show Arrows
  Marker Scale         1
  Update Interval      0
  Frame Timeout        15
  Frames
    All Enabled
    base
    base_link
    base_link_in...
    flange
    forearm_link
    hand_e_link
    robotiq_cou...
    shoulder_link      ✓
      Parent           base_link_inertia
      Position         0; 0; 0.15185
      Orientation      0.68024; -0.19306; ...
      Relative P...    0; 0; 0.15185
      Relative ...     0.68024; -0.19306; ...
    tool0
    tool0_contro...
    upper_arm_l...
    ur_gripper_ti...
    world
    wrist_1_link
    wrist_2_link

base_link_inertia
Enable or disable this individual frame.
```

```
Type: Orbit (rviz)
Current V...   Orbit (rviz)
  Near Cl...   0,01
  Invert ...
  Target ...   <Fixed Frame>
  Distance     1,28987
  Focal S...   0,05
  Focal S...   ✓
  Yaw          5,86359
  Pitch        0,0503985
  Focal P...   -0.37487; -0.13326; ...
```

```
ROS Time: 1681285567.69   ROS Elapsed: 55.97   Wall Time: 1681285567.72   Wall Elapsed: 55.87
Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.
```

Figure 5.2.5: Shows incorrect visual representation in RViz for rotating the base link in the positive direction

This meant that the rotation axis for the “shoulder_pan_joint” should be changed from being around the z-axis to the y-axis. This was checked for all joints by using the polyscope to move the joint in the positive direction and seeing the RViz visualization corresponded to the changes made. After the changes in the robot’s digital description, the frames have the same position and orientation as per the Universal Robots DH parameters documentation as follows:



```
  ✓ Fixed Frame        OK
Grid                   ✓
  ✓ Status: Ok
  Reference Frame      <Fixed Frame>
  Plane Cell Count     10
  Normal Cell Co...    0
  Cell Size            1
  Line Style           Lines
  Color                160; 160; 164
  Alpha                0,5
  Plane                XY
  Offset               0; 0; 0
TF                     ✓
  ✓ Status: Ok
  Show Names
  Show Axes            ✓
  Show Arrows
  Marker Scale         1
  Update Interval      0
  Frame Timeout        15
  Frames
    All Enabled
    base
    base_link          ✓
    base_link_in...    ✓
    flange
    forearm_link       ✓
    hand_e_link
    robotiq_cou...
    shoulder_link      ✓
    tool0
    tool0_contro...
    upper_arm_l...
    ur_gripper_ti...
    world
    wrist_1_link       ✓
    wrist_2_link       ✓
    wrist_3_link       ✓

wrist_3_link
Enable or disable this individual frame.
```

```
Type: Orbit (rviz)
Current V...   Orbit (rviz)
  Near Cl...   0,01
  Invert ...
  Target ...   <Fixed Frame>
  Distance     1,81218
  Focal S...   0,05
  Focal S...   ✓
  Yaw          5,12358
  Pitch        0,460398
  Focal P...   -0.24519; -0.031015...
```

```
ROS Time: 1681220992.86   ROS Elapsed: 178.83   Wall Time: 1681220992.89   Wall Elapsed: 178.80
Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.
```

Figure 5.2.6: Transforms of the robot after updating the Xacro file

Now getting the current pose of the robot yields the following quaternion pose:

position	Quaternion orientation	Euler orientation
x: -0.456486501867 y: -0.371248955154 z: 0.063223457471	x: 0.506123336572 y: 0.493817479342 z: -0.495291677081 w: 0.504648015837	x: 1.5723022851717363 y: 1.5491269558547986 z: -0.0014243021229880636

This is already closer to [-0.45658, -0.36934, 0.06209, 1.547, 1.514, -0.033] since all signs are correct, although the precision is not very high. This is because the TCP is manually calibrated on the polyscope, but the values do not correspond with the one automatically set in the robot model.

MATLAB is used to create two transforms, one for the TCP of the real robot's TCP and one for the TCP from ROS. To set the position of the real robot's TCP, calibration is done until the position of both transforms relative to each other is zero, meaning that the TCP defined by the polyscope and that in ROS overlap. To set the orientation of the polyscope TCP, it is assumed to have zero angle rotations in the x- y- and y- axes to get the difference in rotations relative to the TCP in ROS. The rotation matrix in the resulting transformation matrix is then converted to Euler representation of the angle and the value is updated in TCP settings in the polyscope. This is done to have the retrieved data from ROS about the robot correspond to the polyscope with high accuracy to a have a point of reference for checking data.

The following TCP values use meters for the position and radians for the orientation.

Old TCP	[0.00012, 0.00125, 0.1459, 0.0068, -0.0084, -1.5316]
New TCP	[-0.00025, 0.00245, 0.14789, 0, 0, -1.5708]

To test if the set TCP corresponds to other robot poses, comparison between the pose given by ROS and polyscope are compared for a random pose.

ROS Pose	[-0.39332, -0.11448, 0.707113, 0.599, -0.853, -0.425]
Polyscope Pose	[-0.39622, -0.11298, 0.70532, 0.600, -0.854, -0.431]

Even though the repeatability of the ur3e is $\pm 0.03mm$ as mentioned in Appendix A: UR3e Technical Details, the difference between ROS pose and the Polyscope pose is in the range of mm. To make up for this, further calibration of the TCP is done to get the translation part of the TCP to be more accurate. The resulting translation part of the transformation matrix is as follows: [-0.00295, -0.000025, 0.00177]

It can be seen that the TCP values are not correct starting from the third decimal place, but trying to make these changes to the TCP on the polyscope result in the following message:

```
High precision values will be rounded to two decimal places
```

It is therefore a robot limitation that the polyscope position and ROS position cannot be the same. To overcome this, an external coordinate system drawn on the table on which the robot is mounted will be used for reference.

To verify if the robot is calibrated correctly, an arbitrary start pose will be chosen. The robot will then be moved 0.025 m in each the x- and the y-axis for multiple iterations to see if the error accumulates or it moves in the wrong direction. The two following python code snippets are used for this purpose, where the test_pose is a geometry_msgs/Pose msg with initial pose corresponding to [-0.22747267772053364, -0.12856574740041, 0.02103415792995783, -0.7078271936535141, -0.7063853803836077, -0.00013022500169780916, 0.0005842485304991218]:

For moving in the negative x-direction use:

```
test_pose.position.x -= 0.025
```

And for moving in the positive y-direction use:

```
test_pose.position.y += 0.025
```

The logging is done for each of the poses the robot is supposed to reach, meaning the test_pose and the pose the robot actually goes to using the fx_get_current_pose() function. This data will then be used for analysis of the difference between the goal pose and the actual pose and calculate the error. Each square on the grid of the robot's coordinate system represents 0.025m, hence the choice of distance to move along the axis for the test. The start pose is as follows:

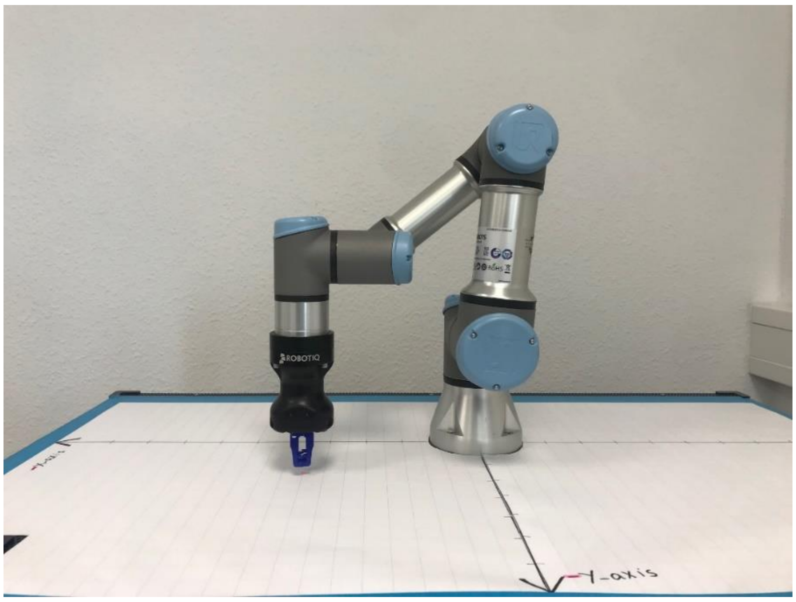


Figure 5.2.7: Robot at the starting pose for the experiment

A mark was made on the coordinate system with each movement in the negative x-direction as follows:

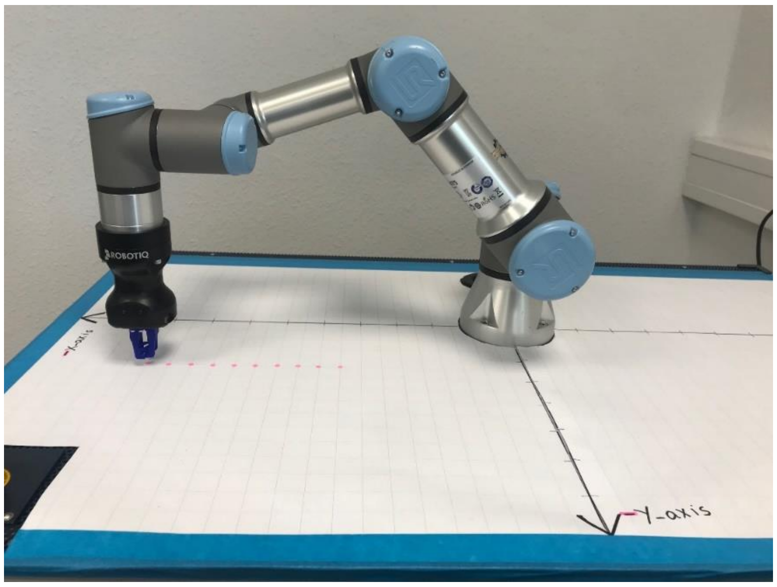


Figure 5.2.8: Markings were made for where the robot stopped after each iteration of the move command along the negative x-axis

The TCP was then moved in 0.025m steps in the positive y-direction to look as follows:

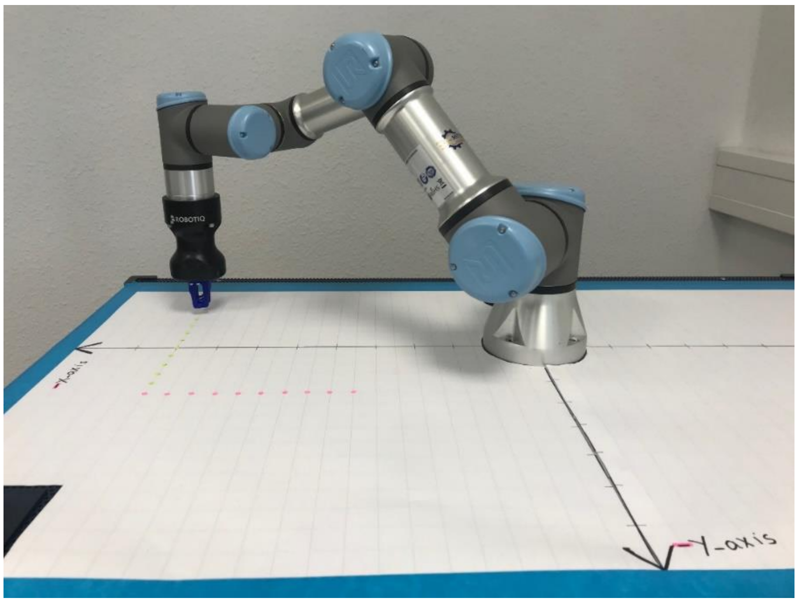


Figure 5.2.9: Markings were made for where the robot stopped after each iteration of the move command along the positive y-axis

The following figure below shows a closer look of the stops the robot made after each iteration of the test move command both along the negative x-axis and the positive y-axis, For more information about the logged data please refer to Appendix F: Testing Linear Movement:

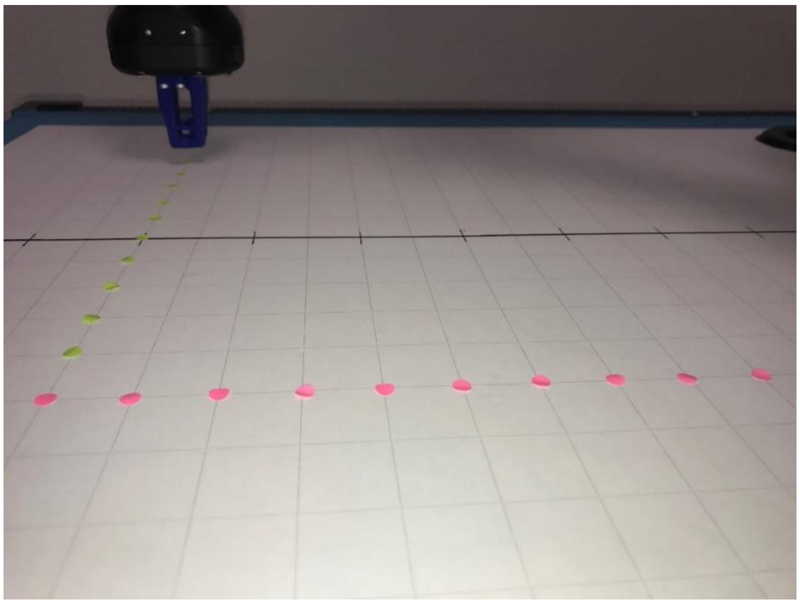


Figure 5.2.10: A close look into the stops made by the robot in both directions for the test move commands

The repeatability of the UR3e is $\pm 0.03mm$ or ± 3.0x 10^{-5} m. To test if the difference between the test and actual points exceeds this resolution the maximum and mean error for the recorded positions from the last experiment were calculated. The maximum error for the recorded poses was recorded to be 9 x 10^{-5} m for the movements along the negative x-axis and 8 x 10^{-5} m along the y-axis test movements. These values are higher than the specified UR3e resolution, so further experiments will be done to improve that error. The mean error is calculated by taking the average of the error of the move commands. The calculated errors are as follows:

Moving in the negative x-direction:

	|Δx|	|Δy|	|Δz|
Maximum Error	8e-05	8e-05	9e-05
Minimum Error	2e-05	1e-05	0
Mean Error	5.75e-05	4.875e-05	4.75e-05

Table 5.2.2: Move command errors in x-direction for default MoveIt! and inverse kinematics solver parameter values.

Moving in the positive y-direction :

	|Δx|	|Δy|	|Δz|
Maximum Error	6e-05	8e-05	8e-05
Minimum Error	1e-05	1e-05	2e-05
Mean Error	3.75e-05	4.875e-05	5.25e-05

Table 5.2.3: Move command errors in y-direction for default MoveIt! and inverse kinematics solver parameter values.

As can be seen all the mean errors exceed the defined UR3e resolution. An investigation will be done to see if these errors can be decreased by changing MoveIt! tolerances or inverse kinematics solver parameters in the next section. But before that, it is important to address the problem that the gripper collided with the table during one of the move commands from the previous experiment. It is therefore important to look for a solution to consider the table in the collision avoidance.

5.2.2. ADDING A VIRTUAL TABLE

When executing the command to move 2.5cm in the x- and y- directions from the previous test, it occurred that the gripper collided with the table. To avoid such collisions with the table, the planning Scene interface can be used to add a box representing the table for collision checking calculations. The following code is used for this purpose:

```
pose_ref = geometry_msgs.msg.PoseStamped()
pose_ref.header.frame_id = robot.get_planning_frame()

pose_ref.pose.position.x = -0.075
pose_ref.pose.position.y = -0.065
pose_ref.pose.position.z = -0.02

pose_ref.pose.orientation.x = 0
pose_ref.pose.orientation.y = 0
pose_ref.pose.orientation.z = 0
pose_ref.pose.orientation.w = 0

scene.add_box("table", pose_ref, (1.15, 0.73, 0.04))
```

Explanation:

The frame of the table is adjusted to make up for the fact that the robot does not lie exactly in the middle of the table. Since the table is flat, the orientation is left at zero. The virtual table is then added into the planning scene under the name “table” and the dimensions of 1.15m x 0.73m x 0.04m

As can be seen, now the manipulator target pose is shown in red when there are collisions caused by the requested movement, and the RViz does not give the option to execute the command.

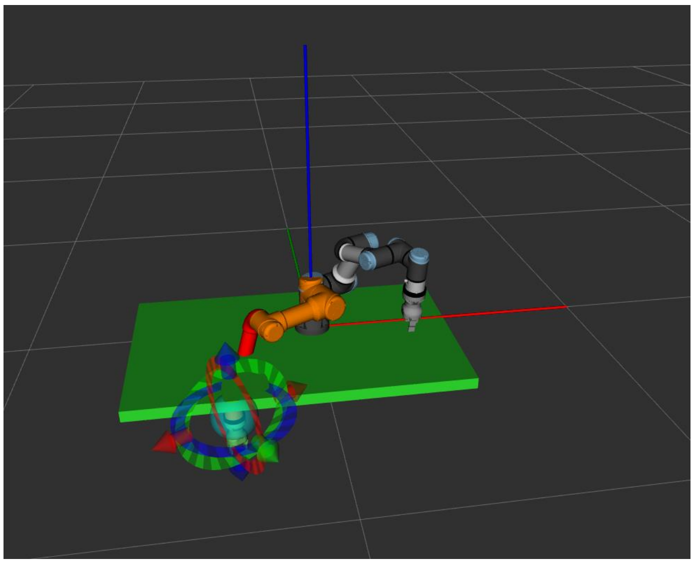


Figure 5.2.11: RViz shows a goal pose that would cause a collision in red

Since the URDF representation and the real robot now match and the robot goes to the target pose, analysis on other aspects of the robot's and the library's performance will be tested in the following sections.

5.3. TESTING AND ANALYSIS

The following part of the testing will use logging and analyzing the logged data to see how the Fluxana Library functions perform by changing different parameters like those of the kinematic solver and MoveIt! as well as configuration parameters like the velocity and acceleration in the joint_limits.yaml file, and to test if changing these parameters improves the precision, accuracy and overall behaviour of the robot during the function execution.

5.3.1. TESTING MOVEIT! TOLERANCE

A given problem is that error between the target pose and the robot's actual pose is bigger than the given precision of UR3e. Since the UR3e precision is given in the technical specifications as $\pm 0.03mm$. To investigate if this error can be decreased by changing the tolerance used by MoveIt!, the following MoveIt! commander command get_goal_position_tolerance() is used to get the current tolerance used. The returned value is the radius of the sphere around the goal pose, where the robot would be considered as having reached its target pose. The returned default value in the study is 0.0001m or 0.1 mm which is higher than the specified UR3e precision. To see if the increased error is from tolerancing in Moveit!, going to a target position from a starting position will be done twice, once with the current tolerance and another with a lower tolerance as shown in the next two tables.

Firstly the going to the test position will be done using the goal position tolerance of 0.0001 which is the value set by default:

Goal Position Tolerance	0.0001		
	X	Y	Z
Test Position	-0.25254188644127	-0.1285600721662	0.02111196684424
Actual Position	-0.25253644021991	-0.1284817769677	0.02104734837189
Difference	5.446221362e-06	7.829519849e-05	6.461847235e-05

Table 5.3.1: Move command errors for goal_position_tolerance = 0.0001

The second resolution 0.00006 is arbitrarily chosen to be higher than the universal robot's precision while considering possible errors in the inverse kinematics calculations. This is to avoid overstepping the specified hardware limitations of the robot and cause unwanted behavior like oscillating between two poses in effort to reach the wanted precision which cannot be reached. Now going to the test pose will be done using the tolerance of 0.00006:

Goal position tolerance	0.00006		
	X	Y	Z
Test Position	-0.2524913981158	-0.1285199482220	0.0210572534056
Actual Position	-0.252502514077	-0.128476577519	0.0210264562451
Difference	1.1115961133e-05	4.33707030908e-05	3.0797160562e-05

Table 5.3.2: Move command errors for goal_position_tolerance = 0.00006

As can be seen the maximum error decreased from 7.829 x10^{-5} m for the tolerance of 0.0001 to 4.337 x10^{-5} m for the tolerance of 0.00006.

To test if the accuracy of the robot has improved, some iterations of move commands are done and the difference between the expected and the actual positions are recorded, and the average of those errors is calculated as follows:

	x-coordinates	**y-coordinates**	**z-coordinates**
Average Error	1.596 x 10^{-5} m	1.729 x10^{-5} m	3.383 x 10^{-5} m

As can be seen the average error for the sample points has decreased along the x-, y-, and z-coordinates in comparison to using the tolerance value 0.0001. For more information on the logged data please refer to the Appendix H: Testing Accuracy for 0.00006 Tolerance.

Trying to make the tolerance too much lower than 0.00006 leads to unpredictable behavior, which could be due to the hardware limitation of the robot. An example of this unpredictable behavior was the robot base link making a full rotation to reach the target instead of moving linearly the 2.5 cm in the negative x-axis direction.

5.3.1. TESTING KINEMATIC SOLVER RESOLUTION

Another source of the error could be the inverse kinematic solver itself. To test if that is the case the parameters for the solver in MoveIt! package will be altered to test if the results have less error. Setting the parameters will be a trade-off between resolution and time, but since accurate positioning in this pick and place application is more valuable than time taken to solve inverse kinematics, the search resolution can be increased to test if the performance improves. The original parameters are as follows:

kinematics.yaml

```
arm:
    kinematics_solver: trac_ik_kinematics_plugin/TRAC_IKKinematicsPlugin
    kinematics_solver_search_resolution: 0.005
    kinematics_solver_timeout: 0.005
```

The kinematics_solver_search_resolution parameter will be set to 0.001 in the kinematics.yaml file for testing as can be seen below:

Old	`kinematics_solver_search_resolution: 0.005`
New	`kinematics_solver_search_resolution: 0.001`

Some iterations of move commands are done and the difference between the expected and the actual positions are recorded to test if the performance was enhanced by changing the inverse kinematics solver resolution parameter. The resulting average error is as follows, where, on average, the errors are below the mentioned UR3e specifications.

	x-coordinates	**y-coordinates**	**z-coordinates**
Average Error	1.761×10^{-5} m	2.301×10^{-5} m	1.697×10^{-5} m

For more details about the logged data for the Appendix I: Testing Accuracy for $kinematics\_solver\_search\_resolution\ = 0.001$.

Comparing the maximum error from the iteration with the $ik\_solver\_precision\ = 0.005$ of 7.05309804e-05m with the error from the iteration with $ik\_solver\ precision\ = 0.001$ of 4.16325099e-05 the error has decreased just by increasing the solver's resolution. These values have been retrieved from Appendix H and Appendix I respectively. Trying to make the resolution to 0.0005 and 0.0001 causes unpredictable behavior, therefore the resolution will be left at 0.001.

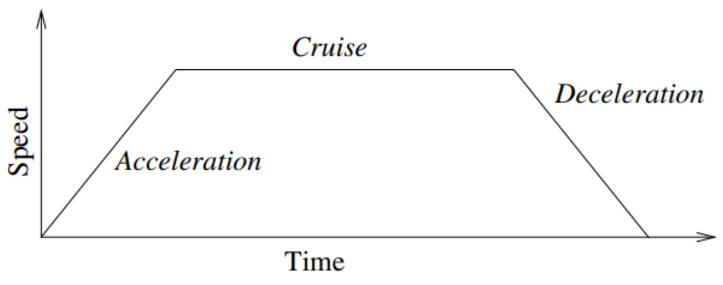
5.3.3. TESTING SETTING VELOCITY USING JOINT_LIMITS.YAML

According to MoveIt! time parametrization documentation, the default velocity used is the one defined in the joint_limit.yaml file. Since the joints are revolute the units are rad/s for the velocity. To test how setting the velocity works, the robot's base will be moved $\frac{\pi}{2}$ radians in joint space from the home position. The time will be recorded before and after executing the commands to calculate the angular velocity. The data will be recorded for different max velocities. Since the base link is the only one moving, only the angular velocity parameter for shoulder_pan_joint will be changed in the joint_limits.yaml file. The resulting data is as follows:

Angular Velocity (rad/s)	Execution Time(s)	Average Angular velocity(rad/s)
1.5	3.634	0.8645
1	4.186	0.7505
0.5	6.339	0.4956
0.3	10.533	0.2983
0.1	31.489	0.0998

Table 5.3.3: Table of the calculated average angular velocities in comparison to the expected angular velocity

As can be seen, the error between the expected velocity and the actual one increases for higher velocity. This is, however, due to not considering the acceleration and the deceleration that takes place. Also, since the acceleration limits are off by default in the file, the actual angular velocity cannot be calculated using kinematic formulas. The following figure shows the speed profile for a UR3e motion as mentioned in the user manual, where the robot needs to accelerate until it reaches the maximum velocity then it starts decelerating again. It may be that the robot takes a long time to accelerate to the max velocity relative to the distance travelled, which is why the average velocity and the maximum velocity have such a big difference:



5.3.1: Speed profile for a UR3e motion [85].

However, the calculated average angular velocity and the actual value become closer as the value gets smaller, this is because the robot takes less time to accelerate and decelerate relative to the distance travelled, therefore, average velocity and actual velocity do not vary greatly. It is concluded that the angular velocities for each of the joints can be set through the joint_limits.yaml file. One way to set joint velocities is by setting the velocity parameter for each joint using the ROS parameter server. Another way is to set the velocity by using a scaling factor between (0, 1]. This can be done using the Move Group Commander functions, through the MotionPlanning tab in RViz or by using ROS' parameter server.

For more control, however, it should be allowed to change the acceleration for the move commands. To allow setting the acceleration, the parameter $has\_acceleration\_limits$ is set to true instead of its default value of false.

The following changes were made to the *joint_limits.yaml* file:

Old	`elbow_joint:`
	`  has_acceleration_limits: false`
	`  max_acceleration: 0`

New	`elbow_joint:`
	`  has_acceleration_limits: true`
	`  max_acceleration: 0.7`

Old	`shoulder_lift_joint:`
	`  has_acceleration_limits: false`
	`  max_acceleration: 0`

New	`shoulder_lift_joint:`
	`  has_acceleration_limits: true`
	`  max_acceleration: 0.7`

Old	`shoulder_pan_joint:`
	`  has_acceleration_limits: false`
	`  max_acceleration: 0`

New	`shoulder_pan_joint:`
	`  has_acceleration_limits: true`
	`  max_acceleration: 0.7`

Old	`wrist_1_joint:`
	`  has_acceleration_limits: false`
	`  max_acceleration: 0`

New	`wrist_1_joint:`
	`  has_acceleration_limits: true`
	`  max_acceleration: 1.4`

Old	`wrist_2_joint:`
	`  has_acceleration_limits: false`
	`  max_acceleration: 0`

New	`wrist_2_joint:`
	`  has_acceleration_limits: true`
	`  max_acceleration: 1.4`

Old	`wrist_3_joint:`
	`  has_acceleration_limits: false`
	`  max_acceleration: 0`

New	`wrist_3_joint:`
	`  has_acceleration_limits: true`
	`  max_acceleration: 1.4`

Where the 1.4rad /s^2 is chosen based on the default angular acceleration value in the polyscope for joint space move commands. Since the wrist1, wrist2, and wrist3 joints have the same velocity limits, and the base, shoulder, and elbow joints have half that limit, the same is mimicked for acceleration. The updated joint_limits.yaml file can be found in Appendix G: Joint Limits for reference.

5.3.5. TESTING PRECISION

The following experiment is to test the precision with which the robot goes to a specific pose. Here the robot will go from a starting pose to a target pose for 10 iterations, where the current pose of the TCP will be recorded and compared to the original target. This comparison will then be used for analysis. Below are the starting and target poses:

Starting Pose	[-0.227472677721, -0.1285657474, 0.02103415793, -0.707827193654, -0.706385380384, -0.00013022500169, 0.000584248530499]

Target Pose	[-0.22747267772, -0.2785657474, 0.0210341579, -0.707827193654, -0.706385380384, -0.000130225001698, 0.000584248530499]

Ten iterations are done and the robot's pose at the target is recorded. These poses are approximated to four decimal places for visibility and the error is calculated by getting the absolute of the difference between the actual pose component and the expected one.

The maximum and mean errors for each component are then calculated and displayed as follows:

```
Errors in poses:
                   position.x  position.y  position.z  orient.x  orient.y  orient.z  orient.w
Iteration 1:        0.0000      0.0000      0.0000      0.0002    0.0002    0.0003    0.0002
Iteration 2:        0.0000      0.0000      0.0000      0.0002    0.0002    0.0006    0.0000
Iteration 3:        0.0000      0.0000      0.0000      0.0003    0.0003    0.0002    0.0002
Iteration 4:        0.0000      0.0000      0.0000      0.0002    0.0002    0.0001    0.0002
Iteration 5:        0.0000      0.0000      0.0000      0.0001    0.0001    0.0000    0.0005
Iteration 6:        0.0000      0.0000      0.0000      1.4155    1.4130    0.0002    0.0006
Iteration 7:        0.0000      0.0000      0.0000      0.0002    0.0002    0.0002    0.0004
Iteration 8:        0.0000      0.0000      0.0000      0.0000    0.0000    0.0002    0.0004
Iteration 9:        0.0000      0.0001      0.0000      0.0001    0.0001    0.0001    0.0002
Iteration 10:       0.0000      0.0000      0.0000      0.0002    0.0002    0.0003    0.0001


Maximum error:
      position.x  position.y  position.z  orient.x  orient.y  orient.z  orient.w
       0.0000      0.0001      0.0000     1.4155    1.4130    0.0006    0.0006


Mean error:
      position.x  position.y  position.z  orient.x  orient.y  orient.z  orient.w
      0.0000      0.0000      0.0000      0.1417    0.1414    0.0002    0.0003
```

Table 5.3.4: Calculated errors between target pose and actual pose for `kinematics_solver_timeout=0.005`

For more information about the logged data refer to Appendix J: Testing Precision. It can be seen that the orientation components x and y have a big error in Iteration 6. Next It will be investigated if this can be improved by changing the timeout parameter of the inverse kinematics solver since the resolution of the solver was changed in previous tests without considering the consequent change in planning time. To test this, the kinematics_solver_timeout parameter will be changed, and the experiment will be repeated. Attached below are the change made to the kinematics.yaml file

Old	`kinematics_solver_timeout: 0.005`
New	`kinematics_solver_timeout: 0.01`

The same experiment is repeated and the maximum and mean errors for each component is then calculated and displayed as follows:

```
Errors in poses:
                position.x   position.y   position.z   orient.x   orient.y   orient.z   orient.w
Iteration 1:    0            0            0            0          0          0.0004     0.0001
Iteration 2:    0            0.0001       0            0.0002     0.0002     0.0002     0.0005
Iteration 3:    0            0            0.0001       0.0003     0.0003     0.0002     0
Iteration 4:    0            0.0001       0.0001       0.0001     0.0001     0          0.0003
Iteration 5:    0            0            0            0.0002     0.0002     0.0002     0.0005
Iteration 6:    0            0            0.0001       0.0003     0.0003     0.0004     0.0001
Iteration 7:    0            0            0            0.0003     0.0003     0          0.0001
Iteration 8:    0            0.0001       0            0.0002     0.0002     0.0007     0
Iteration 9:    0            0            0.0001       0          0          0.0006     0
Iteration 10:   0.0001       0            0            0.0002     0.0002     0          0.0003


Maximum error:
      position.x   position.y   position.z   orient.x   orient.y   orient.z   orient.w
      0.0001       0.0001       0.0001       0.0003     0.0003     0.0007     0.0005


Mean error:
      position.x   position.y   position.z   orient.x   orient.y   orient.z   orient.w
      1e-05        3e-05        4e-05        0.00018    0.00018    0.00027    0.00019
```

Table 5.3.5: Calculated errors between target pose and actual pose for `kinematics_solver_timeout=0.01`

For more information about the logged data, refer to the Appendix K: Testing Precision for kinematics_solver_timeout: 0.01. Now that the precision and accuracy were analyzed for the move functions, the library will be tested using unit testing in python in the next section.

5.4. TESTING FLUXANA LIBRARY FUNCTIONS

Unit testing leads to better code design that is more easily maintained and is an important tool in code development. Testing code using print statements and running functions to check their outputs is not easy to automate and maintain as the code grows. The use of unit testing is to check that the functions are running correctly independent of the rest of the code. Since each of the functions will have a unique behavior, a unit test for each of the main functions will be created to ensure the correct functionality throughout the code [25][73].

The commands used for testing are as follows:

Function	Description
`assertTrue(x)`	Tests if the argument is true
`assertFalse(x)`	Tests if the argument is false

Testing the function will be split into testing kinematic functions first followed by testing the general functions.

5.4.1. TESTING KINEMATIC FUNCTIONS

In the implementation of the following tests, to make sure the robot has had enough time to reach the target pose or joint values, sleep commands are used. This is because the move functions are executed asynchronously by using the MoveIt! command go(wait = False) with the parameter wait set to False instead of the default value True. This is because using go(wait = True) did not allow for the move command to be stopped mid-execution and one of the Fluxana Library specifications is that there should be a stop function. However, those sleep commands are only a temporary solution, and will be replaced by time out mechanisms in the future to optimize the $fx\_move\_pose()$ and the $fx\_move\_joint()$ functions.

Function	Page
Testing Moving to Joint Values	69
Testing Going to Home Position	70
Testing Moving to a Pose	70
Testing Moving Along an Axis	70
Testing Stopping Movement	70
Testing Setting Acceleration Scale	71
Testing Setting Velocity Scales	71

Testing Moving to Joint Values
The test function is implemented under the name $test\_fx\_move\_joint(self)$ and is used to test the Fluxana Library function $fx\_move\_joint(target, velocity\_factor = 1, acceleration\_factor = 1)$. The function defines three sets of test joint values and calls the function under testing to go to these joint values. The test function uses $fx\_get\_current\_joint\_values()$ function to get the current joint values and tests if the actual joint values differ from the value that the joint is supposed to reach by an arbitrarily low margin of error. If the joint values have been reached the test is passed otherwise it is failed.

Testing Going to Home Position

The test function is implemented under the name $test\_fx\_go\_home(self)$ and is used to test the Fluxana Library function $fx\_go\_home(velocity\_factor = 1, acceleration\_factor = 1)$. The function gives the command to go to home position by calling the function under testing. Like with testing Moving to joint values function, the test function calls the $fx\_get\_current\_joint\_values()$ function to get the current joint values and tests if each of the joint values differs from the value that the joint is supposed to reach by an arbitrarily low margin of error. If the joint values have been reached the test is passed otherwise it is failed.

Testing Moving to a Pose

The test function is implemented under the name $test\_fx\_move\_pose(self)$ and is used to test the function $fx\_move\_pose(target, velocity\_factor = 1, acceleration\_factor = 1)$. The test defines three test poses and goes to them. The test function then uses the $fx\_get\_current\_pose\_rpy()$ to get the current pose and checks if the robot reached the updated pose with an arbitrarily low margin of error. If the pose has been reached the test is passed otherwise the test is failed.

Testing Moving Along an Axis

The test function is implemented under the name $test\_fx\_move\_distance(self)$ and is used to test the function $fx\_ move\_distance(distance\_vector, velocity\_factor = 1, acceleration\_factor = 1)$. The test defines three test poses and goes to them, it then increments the x-, y-, and z- directions by the given distance vector of [0.005, 0.005, 0.005] and moves to the new pose using the function under testing. Like the Testing Moving to a Pose function, the test function then uses the $fx\_get\_current\_pose\_rpy()$ to get the current pose and checks if the robot reached the updated pose with an arbitrarily low margin of error. If the pose has been reached the test is passed otherwise the test is failed.

Testing Stopping Movement

The test function is implemented under the name $test\_fx\_stop(self)$ and is used to test the function $fx\_stop()$. The function goes to home position using the $fx\_go\_home()$ Fluxana Library function, it then moves the base joint 90° in the positive direction and issues a command to stop the move command. If the velocity becomes arbitrarily close to zero the test is passed, otherwise, the test is failed.

Testing Setting Acceleration Scale
The test function is implemented under the name $test\_fx\_set\_acceleration\_scale(self)$ and is used to test the function $fx\_set\_acceleration\_factor(acceleration\_factor)$. The test function defines two arrays, one containing the invalid acceleration scale values $[-5, -1.4, 0, 2, 10]$ and tests the function does not accept those passed values. The other array containing the valid velocity scale values of ten equally spaced array starting at 0.1 and ending at 1 is tested to see if the values are accepted by the function under testing.

Testing Setting Velocity Scale
The test function is implemented under the name $test\_fx\_set\_velocity\_scale(self)$ and is used to test the function $fx\_set\_velocity\_factor(velocity\_factor)$. The test function defines two arrays which are the same testing values as in the testing setting acceleration scale test function. The values are passed to the function to check that the invalid values will not be set and that the valid ones will be set, which in that case, the test passes. Otherwise, it fails.

5.4.2. TESTING GENERAL FUNCTIONS

Function	Page
Testing Releasing Brakes	71
Testing Powering On	72
Testing Powering Off	72
Testing Starting and Stopping External Control	72

Testing Releasing Brakes
The test function is implemented under the name $test\_fx\_brake\_release(self)$ and Is used to test the Fluxana Library function $fx\_brake\_release()$ by calling it. The robot mode is then checked if it has changed to “RUNNING” using the fluxana library function $fx\_get\_robot\_mode()$. If the mode was changed, the test is passed, otherwise the test is failed.

Testing Powering On

The test function is implemented under the name $test\_fx\_power\_on(self)$ and Is used to test the Fluxana Library function $fx\_power\_on()$ by calling it and similarly with the testing releasing brakes function checks if the robot mode, which is retrieved with the help of the $fx\_get\_robot\_mode()$ test function, has been changed to “POWER_OFF”. If that is true, the test is passed, otherwise the test is failed.

Testing Powering Off

The test function is implemented under the name $test\_fx\_power\_off(self)$ and Is used to test the Fluxana Library function $fx\_power\_off()$ by calling it and similarly with the testing releasing brakes and power on test functions checks if the robot mode, which is retrieved with the help of the $fx\_get\_robot\_mode()$ function has been changed to “POWER_OFF”. If that is true, the test is passed, otherwise the test is failed.

Testing Starting and Stopping External Control

The test function is implemented under the name $test\_fx\_external\_control(self)$ and is used to test the Fluxana Library functions $fx\_start\_external\_control()$ and $fx\_stop\_external\_control()$. The test function calls the $fx\_start\_external\_control()$ function and checks that the external control program has been started by checking that the program status is “PLAYING” and that the program running has the name “external_control.urp”. The second part of the test function to stops the program by using the Fluxana Library function $fx\_stop\_external\_control()$ and checking that the program has been stopped checking that the program status is “STOPPED”. The status of whether a program is running or stopped as well as the name of the program is retrieved by calling the helper function $fx\_get\_program\_state()$.

5.5 DISCUSSION

In the preparation for testing and validation section, it was concluded that the Xacro robot description did not correspond to the real robot, therefore, changes necessary were made to get both to match correctly. After correcting the Xacro file, it was noticed that the result for getting the pose for using ROS differed from polyscope, therefore experiments were done to improve result, but was concluded that the polyscope and ROS cannot have the exact result since the polyscope has a limitation on how accurate the TCP can be set.

Following that, tests to measure the error between the goal pose and the current pose revealed that the error was too high when compared to the UR3e precision specification. Some parameters like the MoveIt! pose goal tolerance and inverse Kinematic solver resolution and timeout were changed to decrease the initially measured error, and according to measurements, the results were enhanced.

In the end unit tests were created to test the performance of the Fluxana Library functions. The move command were tested by going to a test goal pose and checking if the robot reached that pose by comparing the actual pose or joint values to the target ones. For testing powering on, powering off, releasing braking of the robot, the corresponding function is called and the robot mode is checked if it changed correctly. However, it was noticed that running start external control function under the name fx_start_external_control() for the first time after starting up the robot causes an error in ROS, but calling the function at other point works correctly. The reason for this is unclear and needs to be investigated further in the future.

6. CONCLUSION AND FUTURE WORKS

In the previous chapters, an introduction to the Fluxana Library was made along with the purpose it is used for. The basic knowledge needed in robot kinematics and ROS for implementing the library was then introduced followed by an explanation of the Fluxana Library functions and how they were implemented. Testing of the functions was then done to see if they perform as expected and if they can be enhanced simply by changing parameters. This section will conclude the study and introduce Fluxana's future works to further automate its processes.

6.1. CONCLUSION

In conclusion, this study aims to build a software library using ROS to automate the loading and unloading operations for Fluxana's Vitriox 4+ machine. This library is divided into functions that are used for moving the robot and those that are used for general functions that are essential for the application, like running external control or releasing the robot brakes upon starting the robot, and more functions that are necessary for operation.

The library is implemented with ROS where MoveIt! and rospy functions were used in the implementation. RViz was also used for the visualization and debugging of the software. The testing is done on a Universal Robots UR3e, where it is divided into testing the performance of some functions from setting the different parameters as well as unit testing main functions of the library.

6.2. FUTURE WORKS

The Fluxana library will be expanded to include gripper functions to be able to fully use it for pick and place applications. Fluxana Library will then be used in the creation of a GUI that will be used by the user to access the different functions of the library. This GUI will be run on a Windows computer since the Vitriox 4+ uses a windows machine as depicted in the following figure, where the functions from the Fluxana library use rospy, MoveIt! Commander python interface functions as well as RViz. This windows PC used by the machine will be set as the client and will be communicating with the robot by first communicating with the ROS master running on a Linux machine. The Universal Robots ROS driver on the Linux machine can then interact with the robot by creating a network that contains the robot and the Linux machine on it.

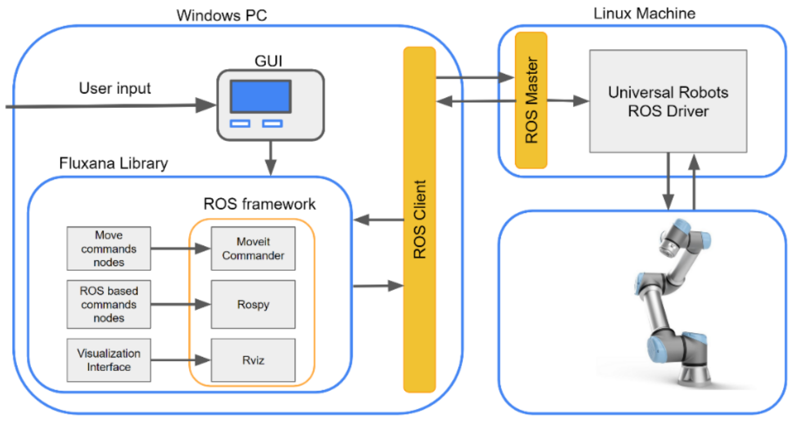


Figure 6.2.1: Future plans fort he UR3e as given by Fluxana's project documentation

Artificial intelligence will be used to enhance the performance of the Fluxana Library. For the loading and unloading of crucibles and glass beads with the UR3e, computer vision and machine learning models will be used to check if the crucibles are placed correctly before starting the machine to avoid safety hazards that can result from the molten sample spilling inside the furnace in the machine during rotation and issue a message for the user in case of an error occurring.

Fluxana, also plans to automate other processes in the future. These projects include powder dosage for fusion processes, transport of crucibles, samples, and glass beads between stations using cobots, and more as can be seen in the figure below, where the arrows indicate the transportation of necessary equipment by the cobot between stations. At each station, the corresponding process will be automated:

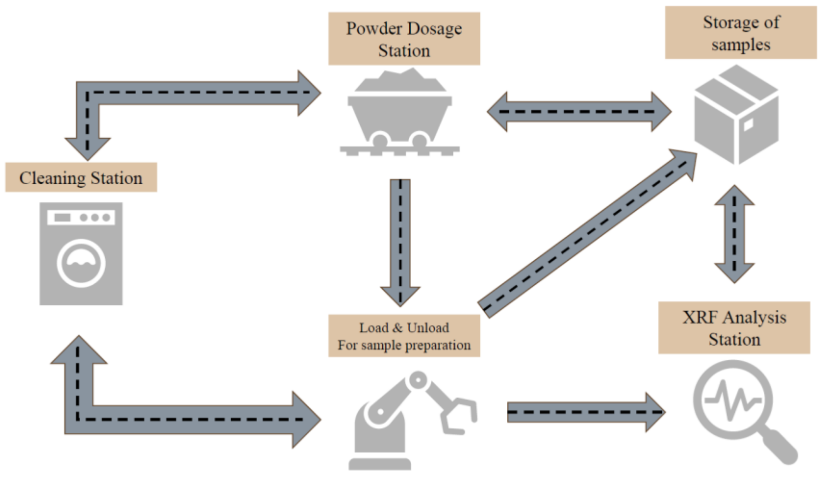


Figure 6.2.2: Fluxana's future projects for automating their processes and the connection between them